University of Oslo
Department of Informatics

Dynamic Modification of Transaction Isolation in the Apotram Transaction Model

Espen Sommerfelt

Siv Ing Thesis

August 2001

Abstract

There is a broad consensus in the database research community that the traditional ACID properties are not suitable for certain application domains. Especially, long-lasting and information sharing transactions are not adequately supported. Several extended transaction models have been presented to deal with these shortcomings. Many of these models do this by relaxing the isolation property. Apotram, presented by Ole Jørgen Anfindsen, is one such model. It allows transactions to customize their degree of isolation. This is achieved by introducing two mechanisms, *parameterized access modes* and *nested databases*. Apotram requires transactions to be able to modify their degree of isolation dynamically. This is achieved by allowing transactions to modify their associated parameterized access modes. This thesis analyzes the consequences of this requirement and points out when these modifications introduce conflicts. First, parameter modification is analyzed in the context of parameterized access modes only, then the integration of access modes and nested databases is investigated. Furthermore, a set of strategies for resolving and avoiding conflicts are introduced and discussed.

Acknowledgements

This master thesis is submitted in partial fulfillment for the *Siv. Ing.* degree in Informatics at the Department of Informatics, University of Oslo (UiO). The work on this thesis was done partly at the Department of Informatics and in the context of the Forrest project at Sun Microsystems Laboratories (SunLabs), Mountain View, California.

I would especially like to thank my advisor, Ole Jørgen Anfindsen for his guidance and help throughout the whole process. He also established the contact with Mick Jordan at SunLabs which resulted in my employment as student intern at SunLabs and the experience of the exciting environment of Silicon Valley. I would also like to thank Mick Jordan for hiring me and Laurent Daynès at SunLabs for valuable discussions during my employment.

Finally, infinite thanks to Marianne for all her love and support.

Contents

Chapter 1

Introduction

1.1 Introduction

This chapter gives some brief background information on transactions, limitations of traditional transactions, and the Apotram transaction model. It also explains what problem this thesis addresses and gives a presentation of the structure of the thesis.

1.2 Background information

1.2.1 An Overview of Transactions

Transactions can be thought of as a unit of operations that define an atomic action. A transaction can be begun, and then completed by either aborting or committing it. If the transaction is aborted then it is as if it never had taken place. Transactions provide *concurrency control* and *recovery*. Concurrency control assures concurrent execution of transactions without introducing inconsistencies. This gives a running transaction the illusion that it has the whole system to itself.

Recovery assures that if a transaction does not succeed for some reason (e.g. hardware or software failure) then the database is restored to a consistent state close to the point of failure. Thus, a failure should never leave the database in an inconsistent state.

Traditionally, transaction are expected to have four properties, called the *ACID* properties [Gra81, HR83]. Briefly explained, they are:

Atomicity. Either all operations of a transaction are executed or none at all.

Consistency. A transaction must be correct, i.e. take the database from one consistent state to another.

Isolation. A transaction should not be able to read intermediate results of other transactions.

Durability. Once completed, the results of the transaction are permanent in the database even in spite of any later failures.

1.2.2 Limitations of Classical Transactions

The flat transaction model was designed for short independent transactions that perform simple state transformations. However, certain application domains have a behavior which can not be adequately supported by flat transactions. Examples of such application domains are CAD/CAM, office automation, publication environments, and software development environments. Transactions in such environments can be very complex, access many data items and live for a long period of time (e.g. hours, days, or months). Long lasting transactions are often called *long-lived*. The flat transaction model is not suitable for such transactions. Firstly, because of the long duration, long-lived transactions are more vulnerable to failures. If such a failure occurs after the transaction has done a possibly significant amount of work, all this work has to be rolled back (i.e. undone). Secondly, to ensure isolation long-lived transactions lock resources for long periods of time and thereby force any competing transactions to wait for the long-lived transaction to commit. Finally, according to [Gra81], the frequency of deadlock increases with the fourth power of the transaction size.

Another limitation of the flat transaction model is that the demand for isolation prevents cooperation between transactions. This poses difficulties in collaborative environments. Transactions are not able to see any ongoing work performed by other transactions. Imagine, for example a software development environment where a team of developers work together on some module. Each transaction represents the work of a developer. Information needs to be shared between the developers to properly integrate their work. Under the flat transaction model the sharing would not be allowed due to the demand for isolation.

This suggests that there is a need to relax isolation to support collaboration between transactions and many of the purposed extended transaction models deal with the shortcomings of flat transactions by doing exactly that. However, there is still a need to preserve isolation in some situations and therefore it would be beneficial if the degree of isolation could be customized. Apotram, briefly summarized in the next section, is one such model that allows transactions to customize their level of isolation.

1.2.3 The Apotram Transaction Model

The Apotram transaction model was defined in [Anf97] by Ole J. Anfindsen. Apotram is an acronym for *application oriented transaction model*. It

deals with the shortcomings of classical transactions by introducing conditional isolation between transactions. This allows transactions to customize the degree of isolation. Only read-write and write-read conflicts can be made conditional, write-write conflicts still conflict unconditionally. This results in a new correctness criterion called *conditional conflict serializability* (CCSR) of which the traditional correctness criterion, conflict serializability (CSR), is a special case. The criterion is implemented by parameterized access modes. A transaction uses parameterized access modes to specify what degree of isolation it wants. A transaction can read uncommitted data from another transaction if their access parameters are compatible. Thus, parameterized access modes realizes collaboration by allowing transactions to share uncommitted data.

Another concept introduced by Apotram is *nested databases*. This allows transactions to recursively create databases. The creating transaction of a nested database can move data items into the database and define which transactions that are allowed to execute within it. Transactions running within a nested database can access its objects. When they commit (or abort) they commit (or abort) to the owner of the nested database, which can accept, deny, or reject the request. Nested databases deal with write-write conflicts by allowing visiting transactions to alternate their write accesses under the control of the transaction owning the nested database. This concept results in the *nested conflict serializability* (NCSR) correctness criterion.

By combining the CCSR and NCSR criteria we get *nested conditional conflict serializability* (NCCSR) which makes the handling of read-write, write-read, and write-write conflicts possible.

1.2.4 The Contributions of this Thesis

One of the requirements of Apotram is the possibility to modify the degree of isolation of transactions dynamically. This is done by allowing the transactions to modify their read and write parameter values during execution. However, these modifications can introduce conflicts. One of the resulting problems is pointed out in [Anf97, page 108]:

> If a data item locked in W(B) mode is read by another transaction in R(A) mode, what should then happen if the first transaction attempts to change lock mode from W(B) to W(C)? Should this be prevented? If not, should the reader be notified? Or perhaps the outcome should depend on whether or not $C \subseteq A$? And should there be rules limiting how lock parameters can change in general?

In general, a conflict can result each time the degree of isolation is made more strict.

This thesis analyses the concept of dynamic parameter modification. Dynamic parameter modification is first investigated under pure CCSR (i.e. nested databases are not allowed.), and then the more complex case of NCCSR is analyzed. The thesis investigates when conflicts result, suggests various strategies to resolve and avoid these conflicts, and gives a discussion of each strategy.

1.3 The Structure of the Thesis

The part of the thesis consisting of chapters 2 through 4 gives some background knowledge required to follow the discussions of this thesis.

Chapter 2 gives an overview of *transactions*. It explains why concurrency control and recovery is needed and how they are ensured by the idea of transactions. Concepts such as the ACID properties, conflict serializability, the two-phase locking protocol, and transaction histories and their properties are explained.

In chapter 3 the concept of *transaction models* is presented. First, the classical flat transaction model and its limitations are discussed. Then, various extended transaction models such as Sagas [GMS87], nested transactions [Mos81], and dynamic restructuring of transactions [KP92] are explained.

A presentation of the *Apotram transaction model* [Anf97] and its properties such as parameterized access modes and nested databases is given in chapter 4.

Chapter 5 is the main contribution of this thesis. It analyses the concept of dynamic modification of parameters and suggest strategies for avoiding and resolving conflicts that can result when applying this concept.

Finally, chapter 6 discusses the results of this thesis, lists the contributions made, and gives some areas for further work.

1.4 Summary

This chapter has given background information and a motivation for this thesis.

Chapter 2

Transactions

2.1 Introduction

> **transaction** /træn'zæk∫n/ *n* **1** [U] ∼ **of sth** the conducting of business: *the transaction of official/routine/government/public business.* **2** [C] a piece of business done: *cheque/credit/cash transactions legal/commercial/property transactions.*
>
> —*Oxford's Advanced Learner's Dictionary*

The transaction is a very old concept. The Sumerians invented writing for transaction processing six thousand years ago [GR93]. The earliest writing that has been found is on clay tablets that recorded the royal inventory of taxes, land, grain, cattle, slaves and gold. This way, scribes kept records of each transaction. One can say that this system was a transactional system:

Database. An abstract system state, represented as marks on clay tablets, was maintained.

Transactions. Scribes recorded state changes with new records (clay tablets) in the database. Today, we would call these state changes *transactions.*

—*Gray, Reuter 1993* [GR93]

The clay tablets represented the real world and made it easy for the scribes to answer questions about the current and past state.

The technology used to record the data evolved over several thousand years through papyrus, parchment and then paper. Then in the the late 1800s, Herman Hollerith built a punch-card computer system to record and report the 1890 United States census. During the first half of the twentieth century the punch-card growth and evolution was heavily fueled by the need

for transaction processing. The systems were primarily used for inventory control and accounting.

Then in the second half of the twentieth century, due to the invention of magnetic storage, batch transaction processing was possible. The first online transaction processing followed batch transaction processing as electronic storage and computer networks. An on-line transaction is the execution of a program that performs a function, usually on behalf of an online user.

The first on-line transaction processing application to receive widespread use was an airline reservations system: the SABRE system [BN97]. It was developed by American Airlines in the early 1960s as a joint venture between IBM and American Airlines. It was one of the biggest computer system efforts undertaken by anyone at that time, and still is one of the largest transaction processing systems in the world.

Table 1 shows some transactional applications with some example transactions.

Application	**Example Transactions**
Banking	Deposit or withdraw money from an account
Securities trading	Purchase 100 shares of stock
Insurance	Pay an insurance premium
Inventory control	Record the arrival of a shipment
Manufacturing	Log a step of an assembly process
Retail	Record a sale
Government	Register an automobile
Internet	Place an order using an on-line catalog
Telecommunications	Connect a telephone call
Military command and control	Fire a missile
Media	Download a video clip

Table 2.1: Examples of transaction processing applications. Taken from Bernstein, Newcomer [BN97]

In the early years, the transaction processing market was primarily driven by large companies needing to support business functions for large numbers of customers. Large TP systems are now becoming even more important as online services become popular on the Internet. But, smaller TP systems will grow as the Internet makes it possible also for small businesses to provide online services [BN97].

2.2 Fundamental Properties of Transactions

A transaction is a set of operations which forms a logical unit of work. The transactions operate on a shared database. The operations of a transaction fall in two groups:

1. Transaction operations: Start, Commit and Abort a transaction.

2. Data operations: Reading and Writing data items.

A transaction is first started, then data operations are performed in the context of the started transaction, finally the transaction is terminated either by a commit or abort operation. No data operations can follow commit or abort. During the execution of an abort all changes made by the transactions are undone/rolled back.

During execution a transaction can go through the following states:

Active. This is the initial state. A transaction is active while it is executing.

Partially Committed. After the commit statement has been executed.

Failed. This state is reached when normal execution can not proceed.

Aborted. After the completion of a rollback of the transaction. The transaction has either aborted itself (suicide) or has been aborted by the system (murder).

Committed. This state is reached when the changes of the transaction have been made durable.

Transactions should possess several fundamental properties. These are usually called the ACID properties [HR83].

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

The acronym ACID was first presented by Härder & Reuter in their article "Principles of Transaction-Oriented Database Recovery" from 1983 [HR83]. The following sections will describe the ACID properties.

2.2.1 Atomicity

Transactions that comply with this property must either execute completely or not at all. It should under no circumstances only execute partially. To give an example of why this property is necessary imagine a transaction that takes $ 100 from account A and then adds it to an account B. This transaction has to be atomic. Either both updates have to be done or none at all. The owner of account A would be very dissatisfied if only the withdrawal of account A would be executed. The $ 100 would be lost.

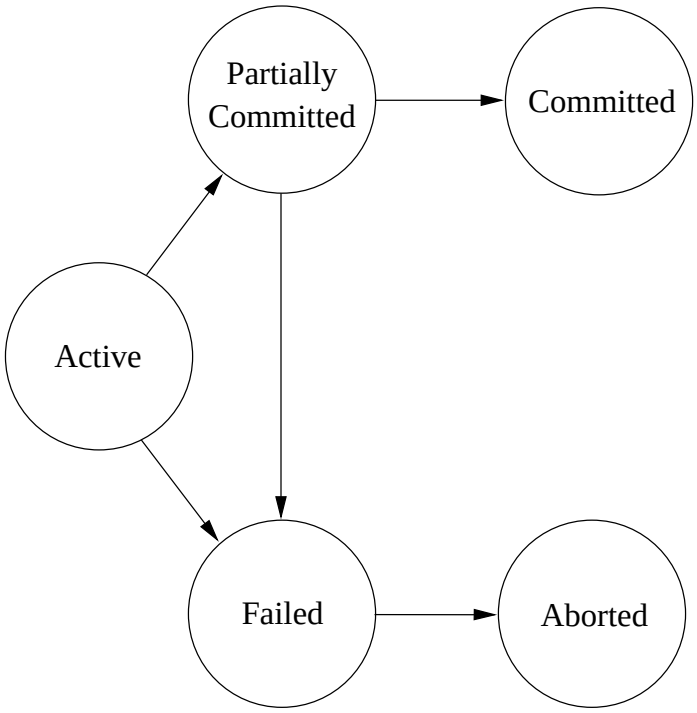


Figure 2.1: State diagram of a transaction

When something goes wrong and the transaction has to be aborted, it is the responsibility of the recovery method to ensure atomicity. All state changes must be undone, and if any other transactions have read data written by the aborting transaction, they too have to be aborted (cascading abort). Section 2.6 will look further into recovery.

2.2.2 Consistency

A transaction should maintain the consistency of the database. When a transaction executes on an initially consistent database the database should also be consistent after the execution. Unlike atomicity, isolation and durability, consistency is a responsibility that is shared between the transaction programs and the transaction processing system that executes those transaction programs. Therefore the programmer of the transaction programs must analyze and test his programs very carefully to make sure that they preserve consistency. During the execution of a transaction the database can be in an inconsistent state a number of times. This fact does not cause any problems because these inconsistencies are not seen by other transactions due to the assurance of isolation and atomicity.

2.2.3 Isolation

Several transactions may run concurrently on a system. If the operations of the concurrent transactions can interleave arbitrarily then their execution can result in an inconsistent database. The isolation property gives each transaction the illusion of having the system entirely to itself. Effects of other concurrent transactions are hidden from the transactions. Section 2.3 will take a look at which problems that can occur when transactions are allowed to interleave arbitrarily.

2.2.4 Durability

Durability requires that all updates that have been done during the execution of a transaction must not be forgotten by the system. This is typically done by storing the updates on some non-volatile device (e.g. a disk) that survives the failure of the system. This guarantees that if something causes the system to fail after a commit, the changes made by the transaction are not lost. This also implies that there is no automatic function to undo the changes of a committed transaction. The only way this can be done is by invoking a *compensating transaction*.

Durability is a important property because transactions usually provide a service that acts as a contract between the users of the system and the service provider. Take, again, the example of a banking system that provides debit/credit transactions. When some amount of money is withdrawn from a customer's account, the bank would not be very pleased if the system forgets the updates of this transaction.

Durability is usually implemented in database systems by logging the changes made by its transactions. The logging is done while the transactions are running. When a transaction is about to commit, the system makes sure that the transaction's log entries are durable (e.g. written to some non-volatile storage). If this is the case, then the transaction has indeed committed and the results are durable. Now, the changes of the transaction are recorded in the log, but they need not be written to the database. So, what would happen if the system fails at this point? The database would have to be repaired using the log. The system checks each entry in the log to see if it has been written to the database. For the log entries where this is not the case, the change of the entry is applied to the database. When this process, called recovery, is completed, the system resumes normal operation.

2.3 Concurrency Control

Concurrency control deals with the coordination of processes that execute in parallel. These processes may access shared data in an interleaved manner which may lead the system into an inconsistent state. This inconsistency

is entirely the result of the way the processes were scheduled and therefore occurs even if every process is coded correctly.

Concurrency problems arise in design of hardware, operating systems, communication systems, real time system, and database systems, among others [BHG87].

In database systems the concurrent processes are represented by transactions. The following section will look at what kind of problems concurrency introduces. These problems are avoided if we do not allow concurrent executions. The transactions will then be executed serially. An execution is serial if, for every pair of transactions, all the operations of one transactions execute before any operations of the other. A serial execution is correct since every transaction involved is correct (by assumption), and serially executing transactions can not interfere with each other. There are two good reasons for allowing concurrency [SKS97]:

1. Throughput. Transactions perform different kinds of actions. One part of the transaction may use the CPU and the other part access I/O devices(e.g. disk, network, keyboard). The CPU and disk can be accessed in parallel by the computer system. By allowing transactions to execute concurrently the parallelism of the computer system may be exploited. Instead of letting the disk be idle during a CPU intensive part of a transaction, another transaction which needs to access the disk may execute concurrently. This increases the throughput of the system.

2. Average response time. The system may run transactions which differ greatly in length. Some may be short while others are long(e.g. a query which has to traverse large portions of the database). If transactions are run serially, short transactions have to wait for long ones to finish. Thus, increasing the average response time.

The motivation for allowing concurrency is essentially the same for database systems as for using multiprogramming in operating systems [SKS97]. Concurrency improves performance but forces implementation of concurrency control schemes to avoid introducing inconsistency. Only in the simplest systems is serial executions a practical way to avoid interference [BHG87].

2.3.1 Problems introduced by concurrency

This section will take a look at four problems which are introduced when transactions are allowed to interleave in certain ways.

The Lost Update Problem

This problem occurs when two concurrent transactions access the same state in a way that causes some update to be lost. Imagine two transactions T_1 and T_2 that both withdraw some money from an account A.

One can see the occurrence of the lost update problem if the two transactions T_1 and T_2 interleave in the manner shown in figure 2.2. The update of transaction T_1, is overwritten by T_2, hence lost.

T_1	T_2
`X:=read_balance();`	
	`X:=read_balance();`
`X:=X-N;`	
	`X:=X-M;`
`write_balance(X);`	
	`write_balance(X);`

Figure 2.2: The lost update problem

The Temporary Update/Dirty Read Problem

This problem happens when one transaction T_1 updates a state and then fails for some reason. If some other concurrent transaction T_2 read the state before it was changed back to its initial value, an incorrect state has been read. The calculations and changes made by T_2 are based on that incorrect value. See Figure 2.3.

T_1	T_2
`X:=read_balance();`	
`X:=X-N;`	
`write_balance(X);`	
	`X:=read_balance();`
	`X:=X-M;`
	`write_balance(X);`
T_1 fails and must undo its changes, but T_2 has already read the inconsistent value.	

Figure 2.3: The temporary update/dirty read problem

The Incorrect Summary problem

If one transaction, T_2, is calculating an aggregate summary function while other transactions are updating some of the values T_2 is using, then it can happen that T_2 reads some values before they are updated and some after. This will cause T_2 to result with an incorrect summary. See figure 2.4.

T_1	T_2	
	`sum:=0;`	
	`read_item(A);`	
	`sum:=sum + A;`	
`read_item(X);`		
`X:=X-M;`		
`write_item(X);`		
	`read_item(X);`	
	`sum:=sum + X;`	⟵ Here T_2 reads `X` after `M` has been subtracted by T_1 but before `M` has been added to `Y`, which gives an incorrect summary in T_2.
	`read_item(Y);`	
	`sum:=sum + Y;`	
`read_item(Y);`		
`Y:=Y+M;`		
`write_item(Y);`		

Figure 2.4: The incorrect summary problem

The Phantom Problem

So far we have only concerned ourselves with read and write operations. Some transactions must also be able to *insert* data into the database, not only modify existing data. Now, imagine that one transaction, T_1, is calculating the sum of all accounts and another transaction,T_2 is inserting a new account. With two transactions only two serial schedules are possible, either $T_1 \rightarrow T_2$ or $T_2 \rightarrow T_1$. In the first schedule, T_1 does not include the new account in the sum, but in the second it is included. Please notice that in schedule 1, T_1 and T_2 do not access any data in common, so one could wrongly think that their operations could be arbitrarily interleaved without introducing a conflict. But, they *do* conflict on a *phantom*, namely the newly created account. So the ordering of the operations affects the result, and as we will see in the next section, this means that the operations conflict.

2.3.2 Conflicting Operations

The reason why these problems occur is due to the ordering of *conflicting operations*. Two operations are said to conflict if their ordering affects the result. This happens if they operate on the same data and one of them is a write. A $read(x)$ operation conflicts with a $write(x)$ and a $write(x)$ operation conflicts with both $read(x)$ and $write(x)$. For example, the result of a $read(x)$ depends on whether the read precedes or follows a $write(x)$ operation. The result of two writes depends on which write was processed last.

2.4 Serializability Theory

To implement isolation and making sure that resulting transaction schedules do not leave the database in an inconsistent state, one has to know which schedules will preserve consistency. *Serializability theory* is a mathematical tool that allows us to prove if a schedule preserves consistency. A concurrent execution of transactions results in a transaction history. The history is said to be serializable if it represents a serializable execution. Informally an execution is serializable if it has the same effects on the database as a serial execution. Because a serial execution assures consistency by not having any interleaving of transactions, a serializable execution also preserves consistency. In this section serializability will be defined and two types of serializability will be explained.

Transaction Histories A history is a model of the execution of a set of transactions. The history consists of the operations of the transactions and how they are ordered. Since some of the operations may execute in parallel the ordering is a partial order. Only non-conflicting operations may be executed in parallel. Because the ordering of conflicting operations determines the result of the history, one has to know in what order they were executed. Each transaction may either have an abort or commit operation. After an abort or commit no operations can perform on behalf of that transaction. A history where all included transactions either have committed or aborted is called a *complete history* [BHG87]. A complete history is *serial* if, for every pair of transactions, all the operations of one transactions execute before any operations of the other. There are $n!$ possible serial histories over a set of n transactions.

Equivalence of histories To be able to define serializable executions we will have to define what it means that two histories are equal.

Definition 1 *[Conflict equivalence of histories] Two histories H and H' are defined as* conflict equivalent *if [BHG87]*

1. *They consist of the same set of transactions and have the same operations.*

2. *They order conflicting operations of non-aborted transactions in the same way. That is, if p and q are conflicting operations and p occurs prior to q in H then p has to occur prior to q in H' as well.*

In the following example T_1 can be thought of as a transfer from account x to y and T_2 as a deposit into account x:

1. $T_1 = r_1[x]r_1[y]w_1[x]w_1[y]c_1$

2. $T_2 = r_2[x]w_2[x]c_2$

Here r[x] and w[x] mean that the transaction respectively reads and writes the data item x. The calculations on the data items are not shown because they do not affect the consistency of the execution of the transactions. We assume that the operations that can be performed by the transactions do not make the transaction able to communicate by other means than by reading and writing data items. If such communication was allowed then the concurrency manager could not determine the dependencies between the communicating transactions.

Consider the following three possible histories including the two transactions:

1. $H_1 = r_1[x]r_1[y]w_1[x]w_1[y]c_1r_2[x]w_2[x]c_2$

2. $H_2 = r_1[x]r_1[y]w_1[x]r_2[x]w_1[y]w_2[x]c_1c_2$

3. $H_3 = r_1[x]r_1[y]r_2[x]w_2[x]w_1[x]w_1[y]c_1c_2$

Note that H_1 is the serial execution $T_1 \rightarrow T_2$. The pairs of conflicting operations between the two transactions are $(r_1[x], w_2[x]), (w_1[x], r_2[x]), (w_1[x], w_2[x])$. The order of these conflicts in the three example histories are:

1. $H_1 : (r_1[x], w_2[x]), (w_1[x], r_2[x]), (w_1[x], w_2[x])$

2. $H_2 : (r_1[x], w_2[x]), (w_1[x], r_2[x]), (w_1[x], w_2[x])$

3. $H_3 : (r_1[x], w_2[x]), (w_2[x], r_1[x]), (w_1[x], w_2[x])$

In H_1 and H_2 the conflicts are ordered in the same way and by definition of equivalence of histories, H_1 and H_2 are equivalent. In H_3, however, the second pair of conflicts is ordered differently. Hence, H_3 is not equal to neither H_1 nor H_2. H_3 is in fact an example of the lost update problem. T_2 overwrites the results of T_1 on x.

This definition of equivalence of histories is based on the ordering of conflicting operations included in the history. Later, in section 2.4.1, we will look at a different definition of equivalence called view equivalence.

Serializable histories Now that we know what a serial history is and what it means that two histories are equivalent, we can understand the definition of serializability:

Definition 2 (Serializability) *A history H is serializable if and only if it is equivalent to some serial history H_S.*

As mentioned earlier there is more than one way to define equivalence of histories; conflict equivalence and view equivalence (see section 2.4.1). Based on whether conflict or view equivalence is used, two forms of serializability are possible, *conflict serializability (CSR)* and *view serializability (VSR)*, respectively. In the rest of this thesis I will let equivalence and serializability mean conflict equivalence and conflict serializability, unless otherwise stated.

In the example above we saw that H_2 is equivalent to the serial history H_1. By the definition of serializability we now know that H_2 is serializable.

Remember that there are $n!$ possible serial histories when n transactions are involved. It gets rather inefficient to find an equivalent serial history to a specific non-serial history when n is moderately large. Another way to test if a history H is serializable is to derive and analyze the *serialization graph*(SG) for H, denoted SG(H). This graph is derived by creating a node for every committed transaction involved in H and creating an edge from T_i to T_j, $i \neq j$, if there is an operation in T_i which precedes and conflicts with one of T_j's operations. An edge in the graph can represent more than one pair of conflicting operations.

The serialization graphs for the histories of the example above are shown in figure 2.5.

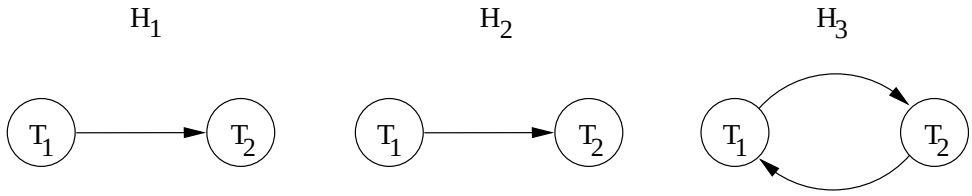


Figure 2.5: Serialization graphs for H_1, H_2 and H_3

The single edge $T_1 \rightarrow T_2$ in the histories H_1 and H_2 represents all three conflicts between the two transactions. The edge $T_2 \rightarrow T_1$ in H_3 exists because of the $(w_2[x], r_1[x])$ conflict ordering.

Each edge in SG(H) means that at least one of T_i's operations precedes and conflicts with one of T_j's. From the definition of conflict equivalence this suggests that if an equivalent serial history exists T_i must precede T_j. If a cycle is present in the graph, an equivalent serial history cannot exist. Let us say that there exists a cycle in H containing the nodes T_i and T_j. In

an equivalent serial history T_i precedes T_j which in turn precedes T_i. This is impossible.

This argument is formalized in the fundamental serializability theorem [BHG87]

Theorem 1 (The Serializability Theorem) *A history H is serializable iff*[1] *SG(H) is acyclic.*

The history H_3 contains a cycle, and therefore is not serializable.

Given an acyclic serialization graph, SG(H), all equivalent serial histories can be derived by topologically sorting SG(H). Our example histories H_1 and H_2 have only one equivalent serial schedule: $T_1 \rightarrow T_2$.

2.4.1 View Serializability

Another definition of history equivalence is called *view equivalence*. This definition of equivalence is less restrictive than conflict equivalence.

Definition 3 (View Equivalence) *Two histories H and H' are view equivalent if the following conditions hold:*

1. *They consist of the same set of transactions and have the same operations.*

2. *For any committing pair of transactions T_i and T_j, if T_j reads a data item x from T_i in H then T_j must also read from T_i in H'.*

3. *If T_i is the last transaction to write a data item x in H, then T_i must also be the last transaction to write x in H'.*

From this definition of history equivalence we get *view serializability.*

Definition 4 (View Serializability) *A history H is* view serializable *iff it is view equivalent to some serial history.*

The definition of view serializability is less restrictive when a value is written that is independent of all previous reads of that transaction. These writes are called *blind writes* [EN94]. Histories that are view serializable but not conflict serializable always contain blind writes [SKS97].

Although view serializability is less restrictive than conflict serializability it is not used in commercial database systems. Testing for view serializability has been shown to be NP–complete, which means that it is highly unlike that efficient algorithms for this will be found. For more information on VSR please consider [BHG87, page 38].

[1] iff reads if and only if.

2.5 Enforcing Serializability

This section will look at the most commonly used concurrency control scheme, *the two-phase locking protocol*. Several other schemes exist:

- Timestamp Ordering (TO).
- Serialization Graph Testing (SGT).

Timestamp ordering and serialization graph testing are theoretically interesting, but hardly used in commercial systems. For a detailed description of these concurrency control schemes see [BHG87, chapter 4].

2.5.1 Schedulers

The concurrency control scheme is enforced by the scheduler. The scheduler receives operations from the transaction manager. It can respond to an operation in three ways:

1. Immediately schedule it.
2. Delay it.
3. Reject it. This causes the transaction that issued the operation to abort.

Based on how schedulers react to operations they can coarsely be divided into two categories: *conservative* and *aggressive*. Conservative schedules tend to delay operations. This gives the scheduler greater freedom to later rearrange the operations to ensure serializability. Aggressive schedulers try to avoid delaying operations by scheduling them as early as possible. The scheduler reduces the ability to rearrange conflicting operations later and may only be able to ensure a serializable execution by aborting one or more transactions.

The performance of the scheduler type depends on the environment in which it is used. Aggressive schedulers may perform better under situations where concurrent transactions rarely conflict. The conservative scheduler might unnecessarily delay operations under this situation. If, on the other hand, concurrent transactions often do conflict then an aggressive scheduler would often have to abort transactions to ensure serializability. The conservative scheduler would more often be able to achieve serializability without aborting transactions.

2.5.2 The Two-Phase Locking Protocol

The oldest and most widely used concurrency control algorithm is *locking* [Tan92]. The concept is simple. Each data item has a lock associated

with it. A transaction can either acquire a lock or release a lock it is already holding. If a transaction T_i wants to use a resource x it first has to aquire a lock on x. If x is not already locked T_i becomes the owner of the lock on x. If, on the other hand, x is locked, no other transaction may access it. If x was already locked by some transaction T_j then it is up to the lock manager to deal with this conflict. The lock manager could choose to deny the request (e.g. by throwing an exception) or block the requesting transaction T_i and add it to a queue. When the owner of the lock on x releases the lock, a transaction T_i is picked from the queue by using some scheduling algorithm (e.g. round robin) and T_i is set to the owner of the lock and revived.

This locking protocol is restrictive because data items only can be locked exclusively. Two transactions can not simultaneously read a data item. More flexibility can be achieved by letting the acquiring transactions specify in what mode they want to lock the item. Transactions can read and write data items, so two lock modes come naturally, read- and write locks.

The algorithm for lock acquisition will now have to consider what lock mode is requested. A request to lock data item x in lock mode m can only be granted if no other transactions hold a lock in mode that conflicts with m. Section 2.3.2 described conflicting operations. Two lock modes on a data item x conflict if one of them is a write. This traditional concurrency control scheme allows multiple simultaneous readers, but only a single writer.

Unrestricted acquisition and release of locks by transactions does not ensure serializability. Cycles can still be created in the serialization graph. Figure 2.6 shows the occurrence of the lost update problem even though locking is used.

T_1	T_2
`rl[x]`	
`r[x]`	
`ul[x]`	
	`rl[x]`
	`r[x]`
	`ul[x]`
`wl[x]`	
`w[x]`	
`ul[x]`	
	`wl[x]`
	`w[x]`
	`ul[x]`

Figure 2.6: The lost update problem when locking is used. The update of T_1 is overwritten by T_2 and therefore lost.

The *two-phase locking protocol* applies some additional restrictions and

guarantees conflict serializability. As indicated by its name the protocol requires that transactions issue their lock and unlock requests in two phases.

1. *Growing phase.* The transaction can aquire locks, but is not allowed to release any lock.

2. *Shrinking phase.* The transaction can only release locks. It is not allowed to aquire any new locks.

The transaction starts with its growing phase. Once a lock is released, the shrinking phase is entered. Once in its shrinking phase the transaction can not acquire any new locks in its remaining execution. Figure 2.7 illustrates the two-phase locking protocol.

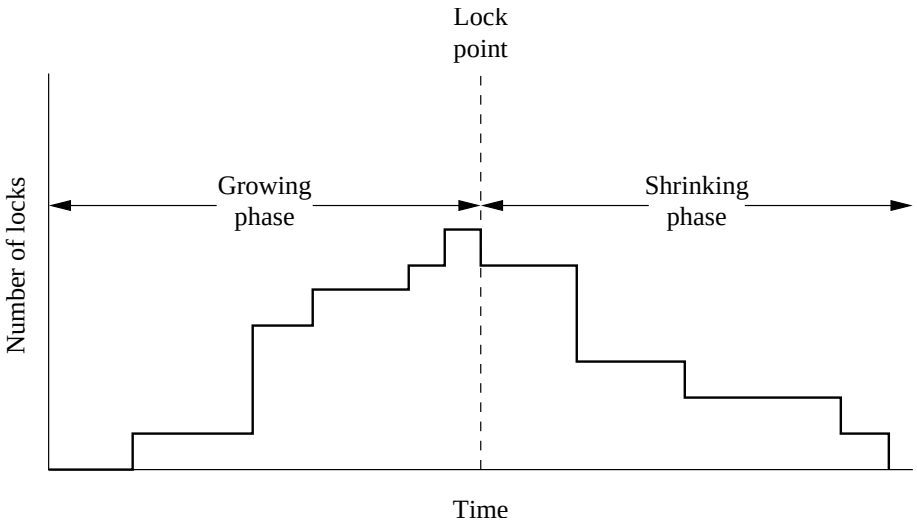


Figure 2.7: The basic two-phase locking protocol (Basic 2PL). The figure has been adopted from [Tan92].

Let us give an intuitive explanation of why 2PL ensures serializability. Recall that a history is serializable if and only if its SG is acyclic. An incoming edge to a transaction's node in the SR may be created when the transaction acquires a lock. An outgoing edge from a transaction's node can only be created if that transaction has released that a lock. Thus, to create a cycle in the SG, some transaction must first release a lock and then later acquire a lock. This is explicitly prohibited by the 2PL protocol. For a more thorough explanation and proof the reader is referred to [BHG87, pages 49–56].

The basic two-phase locking protocol is subject to cascading rollback of transactions. Cascading rollback can be avoided by requiring that write-locks are held until the transaction commits. This version of 2PL is called

strict two-phase locking and assures that no transactions can read results written by uncommitted transactions.

Another variation of 2PL is the *rigorous two-phase locking protocol* which requires that all locks are held until the transaction commits. See section 2.6.1 for more information on rigorous histories.

All of these two-phase protocols are vulnerable to deadlocks. A deadlock occurs when two or more transactions block each other in a way that makes it impossible to continue. Figure 2.8 shows an example of a classic deadlock situation.

T_1	T_2
`rl[x]`	
	`rl[y]`
`wl[y]`	
	`wl[x]`

Figure 2.8: Example of a deadlock. T_1 waits for T_2 to release the lock on y while T_2 waits for T_1 to release the lock on x.

There are various ways of coping with deadlocks.

1. *Time-out.* A transaction that has waited too long for a lock is aborted.
2. *Wait-for graph (WFG).* A graph over which transactions wait for which others is maintained. A cycle in the graph represents a deadlock situation. A transaction participating in the cycle is selected (the victim) and aborted to resolve the deadlock.

It is also possible to modify the 2PL protocols to avoid deadlocks. This can be done by making the transactions predeclare their readsets and writesets. This variant is called *two-phase locking with predeclaring.* A problem with this approach is that the transaction does not always know exactly which items it is going to access. As a solution to this problem, transactions often have to overstate their read- and writesets. For a thorough discussion of deadlocks and 2PL with predeclaring see [BHG87, pages 56–59].

2.6 Recoverability

The recovery system of the database must make sure that software and hardware failures do not corrupt persistent data. This is usually done by restoring the database to a previous consistent sate. All effects of uncommitted transactions should be removed. An executing transaction has the following effects:

1. Effects on data the transaction writes.

2. Effects on other transactions that read from the transaction.

These effects have to be undone when the transaction is aborted.

A transaction has to be rolled back if one of the following situations occur.

- The transaction aborts itself(*suicide*).
- The transaction is aborted(*murder*):
  - Hardware/Software crash.
  - The transaction participates in a deadlock and is chosen as victim.
  - The transaction is scheduled by an aggressive scheduler that can not let the transaction continue because this would break serializability.

It is not always easy to undo all effects of an aborting transaction. Some transactions display output to the user during execution. The user might use this output as input to another transaction. The database system has no possibility to discover this dependency between the two transactions, and the execution of the second transaction might bring the system into an inconsistent state. This way the isolation property has been violated. Some systems solve this by not displaying any output until the transaction has in fact committed all its work. But, this can also lead to a problem. Imagine that the transaction commits successfully and is about to show the user its results. Then, for some reason, the system crashes and the user did not get to see the result. A concrete example would be the withdrawal transaction of an ATM (Automatic Teller Machine). The bank does not want to give the customer his/her money until the transaction has committed. But, if the system crashes between the delivery of the money and the commitment of the transaction, the customer will not receive his money, but the bank will think he did.

These problems are results of non-recoverable actions. The actions can not be undone if the transaction that executes them aborts. The difference between the effect of actions are categorized as [Gra81]:

Unprotected. The action does not have to be considered when aborting a transaction.

Protected. The action has to be undone when a transaction is aborted

Real. The action can not be undone. This category of actions is also known as non-recoverable actions.

2.6.1 Properties of Transaction Histories

Serializability is not enough to ensure executions that preserve consistency. Let us take a look at a transaction history in which the dirty read consistency problem occurs:

- $H_4 = r_1[x]w_1[x]r_2[x]w_2[y]c_2a_1$

Here T_1 updates x and before T_1, aborts its value of x is read by T_2. The serialization graph of H_4 is given in figure 2.9. When analyzing SG(H_4) we find that H_4 in fact is serializable (it contains no cycles) even though it contains a consistency problem. This shows that serialization does not prevent the dirty read problem. Now, because T_2 read from the aborted T_1 it too has to be aborted, a cascading abort. Remember that the durability property of ACID guaranteed that committed transactions are durable. Therefore the durability property forbids the rollback of the committed T_1 transaction. Thus, the history H_4 is not recoverable.

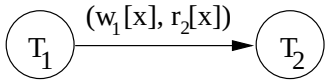


Figure 2.9: The serialization graph of a history that contains a dirty read

A history H is **recoverable** (RC) if no transaction T in H commits until all transactions it read from have committed. A transaction T_i is said to read from T_j in a history H if

1. $w_j[x]$ precedes $r_i[x]$
2. T_j does not abort before $r_i[x]$
3. If another transaction T_k writes x after T_j then it aborts before T_i reads x.

From this definition of recoverable histories we see that H_4 is not recoverable since T_2 reads from T_1 and T_2 commits before T_1.

In recoverable histories no committed transaction will ever have to be rolled back. Notice that if a transaction T_i aborts then all transaction that have read from T_i must be aborted as well. Thus, a single transaction's abort may lead to the abortion of several transactions that may have performed a significant amount of work. This phenomenon is called *cascading rollback*. As an illustration, consider the following example:

- $H_5 = r_1[x]w_1[x]r_2[x]w_2[y]r_3[y]a_1$

T_3 reads from T_2 which in turn reads from T_1. Because T_1 aborts and T_2 reads from T_1, T_2 has to be aborted. T_3 must also be aborted because it reads from T_2.

Cascading aborts are not desirable because they require significant bookkeeping to know which transactions read from which others and because the abortion of a single transaction can result in the abortion of uncontrollably many transactions that may have performed a considerable amount of work. Database systems are in practice designed to avoid cascading aborts [BHG87].

A history H **avoids cascading aborts**(ACA) if a transaction only may read from committed transactions or itself.

Histories that are ACA are not always enough. The problem involves undoing the writes of aborted transactions. We assume that no transactions read from other uncommitted transactions (ACA). Consider the following execution:

- $H_6 = w_1[x, 3]w_2[x, 1]c_1a_2$

The notation $w_i[x, val]$ means that the transaction T_i writes val into x. The write action of T_2 has to be undone. This can be done by using the concept of *before images*. The before image of a $w_i[x, val]$ operation is the value x had just before the execution of this write. To undo a write operation, the write value is replaced by the before image of the write. In our example the value of x should be set to 3 which was the before image of $w_2[x, 1]$. This simple procedure of undoing aborted writes by using before images does not always work. Consider the following execution where the initial value of x is 1:

- $H_7 = w_1[x, 2]w_2[x, 3]a_1$

The before image of $w_1[x, 2]$ is 1, but the value that should be restored is the value 3 written by T_2. The abortion of T_1 should not have any effect on x because x has been overwritten after T_1 wrote it.

Now consider that T_2 also aborts:

- $H_8 = w_1[x, 2]w_2[x, 3]a_1a_2$

The before image of $w_2[x, 3]$ is 2, the value written by T_1. The correct value should be 1, the initial value of x, because T_1 also aborted. This problem occurs when the before image has been written by an aborted transaction.

These problems can be avoided if no transaction can write a data item x before other transactions that have written x have either aborted or committed. Histories which are ACA and satisfy the above requirement are called **strict** [BHG87].

A **rigorous** (RG) history is a strict history with the additional property of not allowing any transaction to write items read by other transactions

until all reading transactions have either committed or aborted. A two–phase locking protocol where all locks are held until the transaction has terminated produces rigorous histories. Rigorous schedulers have been shown to be useful to ensure global serializability in multi-databases [Anf97].

2.6.2 Relationships between classes of histories

The classifications of histories given in the previous section place increasingly more restrictions on histories. This is illustrated by the following theorem [Anf97].

Theorem 2 $RG \subset ST \subset ACA \subset RC$

Figure 2.10 shows the relationships between the recovery and concurrency control properties of transaction histories. Notice that the relations are proper inclusions. This can easily be proven by giving examples of histories which fall into one set but not into the subset.

Almost all commercial database systems implement schedulers which create histories that are conflict serializable and strict (CSR $\cap$ ST) [EN94]. This is due to the fact that conflict serializability can be efficiently implemented by e.g. using strict two–phase locking[2] and that recovery can be simply implemented by the use of before-images when the histories are strict. Being strict also implies, due to the above theorem, that histories are recoverable and avoid cascading aborts.

2.7 Summary

This chapter has given an explanation of the concept of transactions and presented basic terminology and formalism. Transactions provide concurrency control and recovery. Both these aspects where explained and techniques to ensure them were presented.

[2] A strict two–phase locking scheduler will actually produce histories that are stricter than strict [Anf97, page 11].

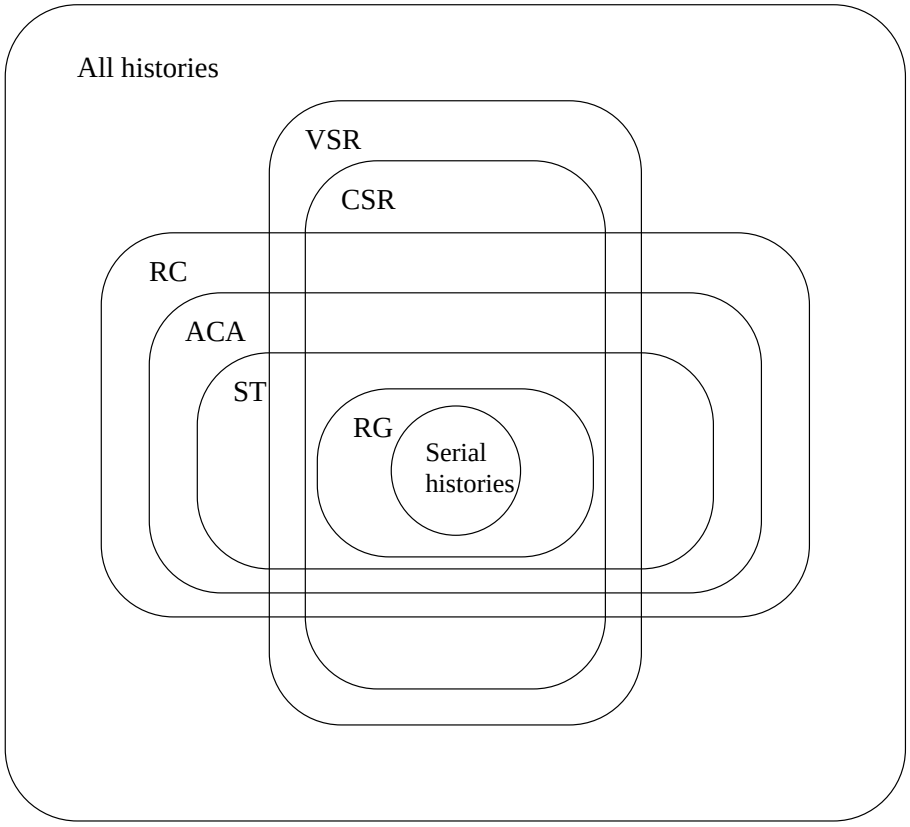


Figure 2.10: Relationships between histories that are CSR, VSR, RC, ACA, ST, and RG

Chapter 3

Transaction Models

3.1 Introduction

This chapter explains the concept of transaction models and gives examples of such models. Its starts with discussing the traditional flat transaction model in section 3.3. Section 3.4 motivates the need for more advanced transaction models and shows that flat transactions give inadequate support for a range of application domains. One type of transactions that are poorly supported by the flat transaction model are long-lived transactions, which are discussed in section 3.5. The Saga transaction model was proposed to handle long-lived transactions and is presented in section 3.6. Section 3.8 introduces the nested transaction model, which is the fundamental basis for almost all extended transaction models. Both the flat and nested transaction models are inspired by the theoretical concept of spheres of control presented in section 3.7.

3.2 Transaction Models

A transaction model specifies what components it is made of, how they are structured, and how they can and must behave. For example, the nested transaction model allows a hierarchical structure of transactions and constrains behavior by requiring that a parent transaction only can commit if all its children have committed (see section 3.8).

The following sections will discuss various transaction models.

3.3 Flat transactions

The flat transaction model is the traditional and simplest transaction model. Almost all existing systems today only support this model. The examples of transactions given in the previous chapter where all of flat transactions.

A transaction starts with a begin and ends with either a commit or abort (see 2.1). A transaction is called flat because there is only one layer of control. Everything between begin and commit/abort occurs on the same level. This means that either all the actions of a transactions will commit or abort. It is not possible to rollback or commit parts of a transaction. As we will see later, there exists a need for a more control of the execution of transactions. This need has lead to several suggestions of extensions of the flat transaction model.

Figure 3.1 shows one of the most commonly used example of a flat transaction: the debit/credit transaction. The flat transaction model was first created for banking applications, which makes it very suitable for these kinds of transactions.

```
procedure DebitCreditAccount( Account account, float amount )
{
   Begin_Transaction();

   account.balance := account.balance + amount;
   if( account.balance < 0 )
   {
     Abort_Transaction();
   }
   else
   {
     Commit_Transaction();
   }
}
```

Figure 3.1: Example of a flat transaction: the debit/credit transaction

3.4 Extended Transaction Models

The flat transaction model is highly suitable for short independent transactions that perform simple state transformations. It was designed with this behavior in mind. However, the complexity of certain application domains results in behavior that the flat transaction model is unsuitable for. There is a broad consensus for this claim in the research community and many extensions of the flat transaction model have been suggested to deal with the requirements of these advanced(non-traditional) application domains. CAD/CAM, CASE, cooperative applications are examples of such advanced application domains.

The following examples show activities that are not supported by the flat

transactions model:

Trip planning. Imagine a system for a travel agency that supports trip planning. During the trip planning the agency can book various activities, like flights, hotels, rental cars and so on. Now, let's say that a fair amount of booking has been done when the travel agent finds out that the last booking has to be canceled for some reason. The scope of rollback in the flat transaction model is the whole transaction. All the work that has been done will unnecessarily be rolled back. It would be useful to be able to do a *selective* rollback. Instead of aborting the whole transaction it should be possible to rollback to a selected position. In section 3.8 we look at how the nested transaction model supports fine-tuning of the scope of rollback.

Bulk update. At the end of a month a bank has to update the accumulated interest of its one million accounts. This large amount of work is performed by a transaction T. Now, imagine that the system crashes for some reason and that T had done the significant amount of work of updating 940,000 accounts. The very undesirable effect of the crash, is that T now has to be rolled back. All the 940,000 updates have to be undone, although they are not invalid. The rollback will probably take the same amount of time as the work already done. A more acceptable behavior would be for the transaction to be able to pick up at the last account successfully updated and continue updating from there.

Unpredictable developments. Take a software development environment where developers can lock files and work on them. Imagine that a developer Bob needs two files to perform his development assignment and write-locks both of them. Bob finishes his work on A and continues his work on B. While doing his changes to B, another developer Alice asks him if she can get access to A. Bob does not need A anymore and wants to be able to release A. If his transaction adheres to the two-phase locking protocol, releasing A would force Bob's transaction into the shrinking phase. Bob would not be able to aquire any new files. In addition, if the system avoids cascading aborts, Bob's transaction would not be allowed to release any files until the transaction commits. Thus, forcing Alice to wait for Bob to finish his possibly large amount of work on B. It would be desirable for Bob to only commit the work on A or to be able to atomically transfer A to Alice's transaction. The atomic transfer solution would be attractive if Bob did not want the outside world to see his changes on B before Alice commits her work. These two alternatives can be realized by using the dynamic restructuring of transactions suggested by Gail Kaiser and Carlton Pu in [KP92].

This example demonstrates that some environments where the actions of a transaction are not foreseeable can not be properly supported by traditional transaction management.

CAD design. A number of engineering teams work in parallel on the design of a new car. Team A works on the engine and team B works on the transmission. Each team's work is done in the context of a transaction, T_A and T_B. During the course of the work the teams have to collaborate to make sure that the engine and transmission actually fit together when the work is completed. But, because of the isolation property of ACID, one transaction can not see the effects of another until it has committed.

This example shows that in some situations it is required that concurrent uncommitted transactions can see each others uncommitted results, and in that way be able to cooperate.

These examples show a number of situations where traditional transaction management does not offer adequate support. The core requirements can be summarized as:

Long-lived transactions. Long-lived transactions are discussed in section 3.5.

Inner-structuring of transactions. Instead of requiring transactions to be flat, some sort of inner structuring should be supported by the transaction model. This structuring can be used to fine-tune the scope of rollback or exploit inner transaction parallelism. The Saga and nested transaction models discussed in section 3.6 and 3.8 respectively, provide structuring of transactions.

Cooperative transactions. To allow transactions to collaborate they must be able to share information. This is explicitly prohibited by the isolation property, which makes it appear to transactions that they are executing alone. This suggests that isolation has to be compromised. The Apotram transaction model presented in chapter 4 supports cooperation.

Serializability is a too strong correctness criterion to support long-lived and cooperative transactions. Many extended transaction models provide support for these transactions by introducing correctness criteria that relaxes isolation/serializability.

The basic idea is that an identified conflict with respect to serializability is not necessarily really a conflict given a specific application's semantics. Take the example of a collaborative document editing application: having one transaction reading a document while another is modifying it does not necessarily represent a conflict in the eyes of the application.

Using the application's semantics to define the concurrency control scheme results in moving some responsibility for the integrity of the data from the database management system to the application. This is the price one has to pay when relaxing isolation. This kind of concurrency control is often called semantic concurrency control.

3.5 Long-Lived Transactions

Long-lived transactions are transactions that last for a significantly long period of time(e.g. days, weeks or months). A long-lived transaction (LLT) has a long duration because it accesses many objects, performs time consuming computations, waits for interaction with humans, or a combination of these factors. As already shown in the section 3.4, these kinds of transactions are not properly supported by the traditional transaction model. The bulk update example showed how vulnerable long-lived transactions are to failures. It is not acceptable to completely rollback a transaction that has performed, say, a week's amount of work. LLTs also have a higher probability of encountering a system failure due to their duration. In addition, a long lived transaction holds locks on resources and thereby prevents other concurrent transactions from accessing these resources during its lifetime. The other transactions may have to wait weeks or even months before they are able to access the locked resources, thus heavily reducing concurrency. This is a consequence of enforcing isolation.

Extended transaction models often provide support for long lived transactions by relaxing isolation. They can make it possible to release resources early by committing parts of a transaction or to allow collaborative sharing of resources between transactions.

3.6 Sagas

Sagas where introduced by Garcia-Molina and Salem in [GMS87] to deal with the problems of long lived transactions. Compared to traditional models, Sagas relax the property of isolation by allowing a Saga to release partial results before it completes.

A Saga is a LLT that can be broken up into a set of subtransactions that can be interleaved with other transactions. A subtransaction represents a part of the work done by the complete transaction. Each subtransaction is a transaction in its own right. The effects of a committed subtransaction will be globally visible even though the entire Saga may not be completed. Thus, the execution of the Saga is not isolated. The compound execution of the subtransactions of a Saga should be atomic. If any subtransaction fails then all completed subtransactions should be compensated for. The compensation is made possible by providing each subtransaction with a *compensating*

transaction. The compensating transaction undoes the semantic effects of its subtransaction. Because, the execution of the Saga is not isolated, other transactions could have read the results of the subtransactions before they were undone by its compensating transaction. It is therefore necessary to assure that the reduced isolation does not introduce inconsistencies.

In section 2.6 we saw that in real life some actions, called real actions, can not be compensated (e.g. drilling a hole). In these cases, Sagas are inapplicable.

The idea of incorporating compensating transactions into the transaction model is the main contribution of Sagas [ELMB92].

3.7 Spheres of Control

This section will describe a concept which was the inspiration for the flat and nested transaction models: Spheres of Control.

The notion of spheres of was presented in [Dav78]. The idea is general and powerful, but has never been fully formalized. A system that fully supports spheres of control would probably be hard to implement due to the generality.

For a system to be able to use the concept of spheres of control it has to be structured as a hierarchy of abstract data types. The abstract data type, or ADT, hides its internal effects from the surrounding environment in case it has to revoke the result for internal reasons. The operation is first externalized when the ADT returns the result through its interface. Thus, each invocation of an ADT is an atomic action from the callers point of view.

When ADTs are organized in hierarchies, lower level ADTs are used to compose higher level complex ADTs. It is not always desirable to make the results of the lower level ADTs globally available. Consider the operation representing a transfer from one bank account to another. This can be used by representing two ADTS: the Bank and the Account ADT. Assume that the account ADT has two atomic operations: deposit and withdraw. The transfer of $100 from an account A to another account B can be implemented by first issuing an withdraw($100) from account A, followed by an deposit($100) on account B. The complete transfer operation should be atomic. If each operation on the account ADT is externalized, then the outside world would be able to see the results of each of the operations. Imagine now that the deposit operation fails due to some error. A process that used the externalized results of the withdraw operation would now base its operations on erroneous data. To avoid this problem and assure atomicity, it is possible to dynamically create new spheres of control that contain the commitment of data. A *dynamic sphere of control* would be created which contains the effects of the deposit and withdraw operations. These dynamic spheres are determined by consistency constraints and dependen-

cies on shared data, messages or other objects.

This results in two varieties of spheres of control.

1. *Static SoCs.* These SoCs are created by the structuring of the abstract data types.

2. *Dynamic SoCs.* These SoCs are dynamically created to contain the commitment of shared data.

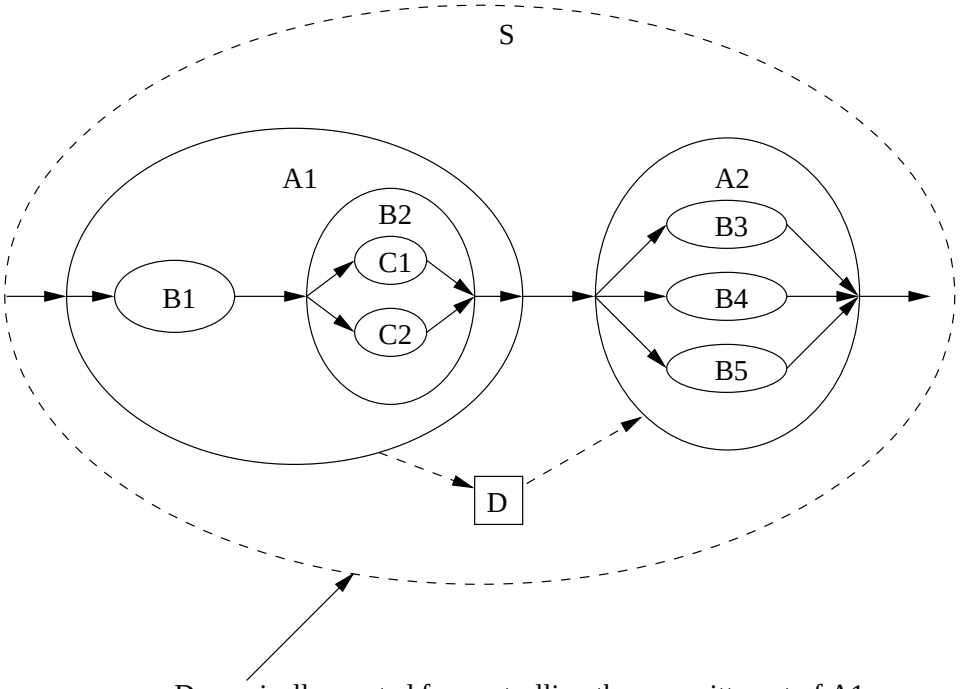


Figure 3.2: Spheres of control

Figure 3.2 shows these two kinds of spheres of control. The spheres with solid lines represent the abstract data types and the sphere with a dotted line is a dynamic sphere. The arrows show an execution through the ADT hierarchy. A1 and A2 are executed sequentially, while e.g. C1 and C2 are executed in parallel. A1 wants to retain some of its effects from being externalized and therefore creates the dynamic SoC S. Now, A2 can start working on the data of A1. S contains the joint action of both A1 and A2 and can terminate when all other processes which depend on A1 have terminated.

The spheres of control concepts suggest that the complete execution history with dependencies and values associated should be recorded. This implies that data items would never be changed, instead a new version would

be created. Another consequence of this approach is that there is not really any need to keep a database. The log will contain all the information.

The reason why this extensive history has to be recorded comes from the demand for recovery. Imagine that some SoC B encounters an erroneous data item. It now has to trace back and find the SoC that created the error. SoC B finds out that the faulty data was created by SoC A. The next step now will be to trace forward and do recovery for all processes that have become dependent on data produced by SoC A. The concept of spheres of control does not provide any help to do this recovery. All the recovery steps are completely application dependent.

The only thing that a SoC system provides is a history of data versions and dependencies. The application has to deal with anything else.

3.8 Nested Transactions

3.8.1 Introduction

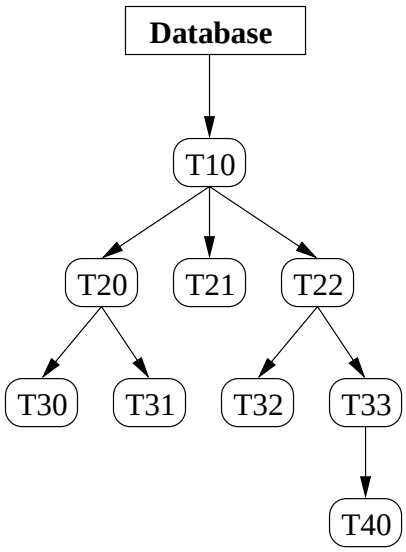
The nested transaction model was introduced by Moss in [Mos81]. The idea of nested transactions seems to have originated with the spheres of control concept of Davies [Dav78](See section 3.7). He defined a nested transaction hierarchy as a nested collection of spheres of control where the outmost sphere was called the top-level transaction. The top-level transaction was the interface to the outside world.

The nested transaction model allows transactions within transactions. A nested transaction is a transaction that contains *subtransactions*. The subtransactions may themselves be nested. This nesting of transactions results in a hierarchy of transactions. A graphical illustration of a nested transaction hierarchy and a description of the terminology is given in figure 3.3.

Nested transactions have at least two important advantages over traditional flat transactions. First, they allow potential internal parallelism of transactions to be exploited (See section 3.8.3). Second, they make it possible to define the scope of rollbacks by allowing subtransactions to fail independently.

The nested transaction model is the fundamental basis of all advanced transaction models which are proposed in the literature [US92]. They differ in the constraints they put on how transactions are structured and how they interact with each other. Despite this common use of the nested transaction model very few commercial systems support the model even today.

Consider once more the *trip planing* example. This problem can now be represented by letting the complete trip planning be the top level transaction and each individual reservation a subtransaction of the top level transaction. Now, if one of the individual reservations fails, only the respective subtransaction has to be rolled back. The other transactions are not affected. Remember that when using the flat transaction model, all the work had to be



root or top level transaction: root of the transaction hierarchy (Here T10).

superior, ancestor: each transaction which is a part of a given transaction up to the database. The ancestor includes the given transaction itself. $superiors(T33) = \{T22, T10\}$. $ancestors(T33) = \{T33, T22, T10\}$.

inferior, descendant: each transaction which is a part of the subtransaction hierarchy of a given transaction. The descendant relation includes the given transaction itself. $inferiors(T22) = \{T32, T33, T40\}$. $decendants(T22) = \{T22, T32, T33, T40\}$.

parent: The immediate superior of a transaction. $parent(T22) = \{T10\}$.

child: The immediate inferior of a transaction. $children(T22) = \{T32, T33\}$.

sibling: Any other child of the parent of a given transaction. $sieblings(T22) = \{T20, T21\}$.

leaf transaction: Transaction which has no inferior. In the above transaction tree $T21$, $T30$, $T32$, $T32$ and $T40$ are all leaf transactions.

non-leaf (or inner) transaction: All transactions which have at least one inferior. $T10$, $T20$, $T22$ and $T33$ are non-leaf transactions.

Figure 3.3: Nested transaction terminology. This figure is adopted from [US92].

rolled back (see section 3.3). In addition, independent transactions could be executed in parallel. We will look further at parallelism in section 3.8.3.

3.8.2 Definition of the Nesting structure

As already mentioned a transaction can start a subtransaction that recursively can create its own subtransactions. The following rules define the relationship between transactions and subtransactions [GR93].

Commit rule. When a subtransaction commits its results are committed to its parent transaction, T_p. These results are only visible to T_p. The transaction completely commits after all its superiors up to the root transaction have committed. After the root transaction has committed the results of T are available to the outside world and made durable.

Rollback Rule. When a subtransaction is rolled back all of its inferior transactions are also rolled back, independent of their local commit status.

Visibility rule. Changes made by a subtransaction are made visible to its parent when the subtransaction commits. Objects held by the parent of a transaction can be made available to its children.

From the outside a top level transaction looks and behaves exactly as a traditional flat transaction. Subtransactions are not fully equivalent to flat transactions. The commit rule states that the results are not made durable until the top-level transaction commits. Therefore, they lack the durability property of the ACID properties. Even though a subtransaction has issued a commit, if any superior of the transaction is rolled back, then the committed subtransaction is rolled back.

Subtransactions are atomic with respect to their parent transaction. Moss defined that only leaf transactions can do actual work on the database. Inner transactions where used to structure and control the work being done by the leaf transactions. Many proposed transaction models that are based on nesting do not apply this restriction.

3.8.3 Parallelism

Besides allowing control over the scope of rollbacks, another benefit of nested transaction is their ability to exploit parallelism. Parallelism can be categorized into two kinds:

1. *Inter-transaction parallelism.* Allows parallelism between transactions.

2. *Intra-transaction parallelism.* Allows parallelism within a transaction.

Recall that from the outside a nested transaction is indistinguishable from a flat transaction. Flat transactions can already be executed in parallel, thus introducing nesting does not increase inter-transaction parallelism. Nested transactions do however support intra-transaction parallelism. By taking a flat transaction and structuring it into a hierarchy in which parts can be executed in parallel exploits intra-transaction parallelism.

Several different policies of intra-transaction parallelism can be realized based on what restrictions are applied [Anf97]:

1. *No parallelism.* Only one member transaction of a nested transaction can be active at any one time.

2. *Parent-child parallelism only.* Only the parent and at most one of its children can be active at any one time. The child can recursively have only one child active. Parallelism is restricted to some hierarchical path. Sibling transactions can not run in parallel.

3. *Sibling parallelism only.* Several children of a parent may be active, but the parent must remain passive until all of them have completed.

4. *Both Parent-child and sibling parallelism.* Parallelism without any restrictions. All members of the transaction hierarchy may execute in parallel.

To avoid trouble caused by the increasingly allowed concurrency of the above policies, concurrency control mechanisms are required.

3.9 Dynamic Restructuring of Transactions

Dynamic restructuring of transactions was proposed in [KP92]. It was purposed mainly for supporting *open-ended applications.* Open-ended applications are characterized by:

1. Uncertain duration (from hours to months).

2. Unpredictable developments (actions can not be foreseen at the beginning).

3. Interaction with other concurrent activities.

Dynamic restructuring is realized by the following provided operations:

- Split-Transaction and Split-Commit-Transaction
- Join-Transaction

The `Split-Transaction` operation splits an ongoing transaction into two serializable transactions and divides its resources between the resulting transactions. The work of a transaction can be divided among several coworkers by splitting it. The syntax of the operation is:

```
Split-Transaction(
A:(AReadSet, AWriteSet, AProcedure),
B:(BReadSet, BWriteSet, BProcedure))
```

`A` and `B` denote the resulting transactions of the split and `AReadSet`, `AWriteSet`, `BReadSet`, and `BWriteSet` are sets of data items accessed in `A` and `B`. `AProcedure` and `BProcedure` are the starting points of execution for `A` and `B`, respectively. The `Split-Commit-Transaction` operation is a variation of the split operation where the new transaction resulting from a split is immediately committed. This operation is useful to release held resources early and thereby increasing concurrency. The `Split-Commit-Transaction` command has the following signature:

```
Split-Transaction(
A:(AReadSet, AWriteSet),
B:(BReadSet, BWriteSet, BProcedure))
```

The `AProcedure` argument is not applicable because `A` will commit immediately after the `Split-Commit-Transaction` operation.

The `Join-Transaction` operation atomically transfers all the resources held by an ongoing transaction into a specified target transaction. The transaction that is joined into the target transaction is dissolved after the join operation and its resources are committed or aborted as part of the target transaction. The operation allows a hand-over of results to a coworker to integrate them into the coworker's ongoing task. The syntax of the join-transaction operation is: `Join-Transaction(S:TID)`. `S` is here the transaction identifier of the target transaction. In order to maintain control over its execution, the target transaction should be able to decide if it allows another transaction to merge with it. These operations are illustrated in figure 3.4.

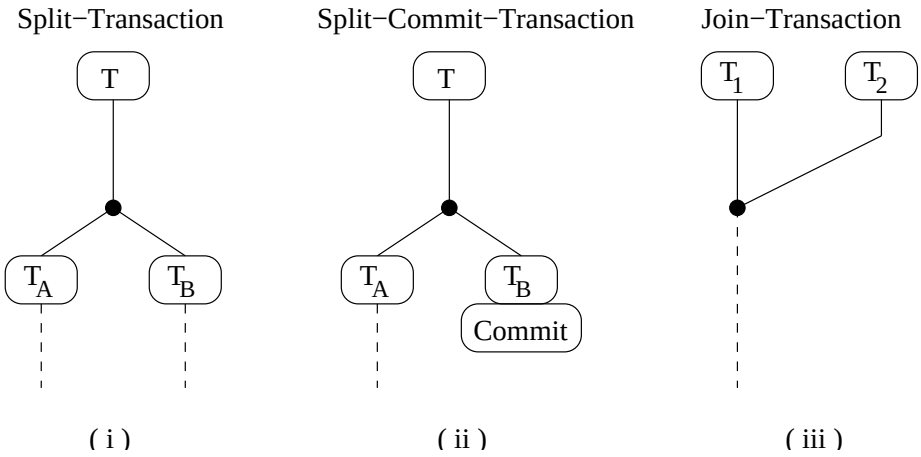


Figure 3.4: Dynamic restructuring of transactions

By combining the split and join operations an atomic transfer of resources can be realized. Imagine that two transactions need to collaborate

by transfering some data from one to the other. This can be done by splitting the data to be transfered to a new transaction and then immediately joining the transaction with the target transaction of the collaboration. This is illustrated in figure 3.5.

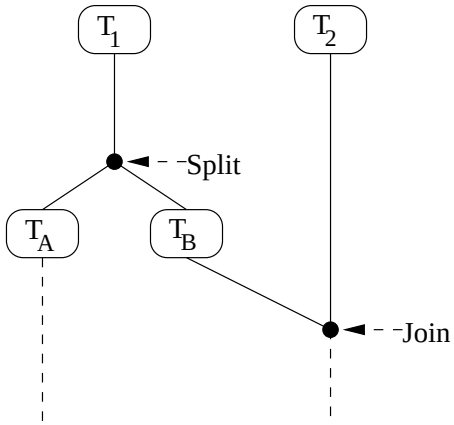


Figure 3.5: Atomic transfer of resources through use of split and join operations

3.10 Summary

In this chapter we have taken a look at the flat transaction model and its limitations. Requirements of advanced applications that are not supported by the flat transaction models were discussed. Long-Lived transactions, Sagas, Spheres of control, nested transactions, and dynamic restructuring of transactions are all extended transaction models that aim at dealing with the shortcomings of flat transactions and were presented through the rest of the chapter.

Chapter 4

Apotram

4.1 Introduction

Apotram is an extended transactional model defined by Ole Jørgen Anfindsen in [Anf97]. The Apotram acronym stands for *Application-Oriented Transactional Model* and indicates that the behavior of the model can be configured by the application. The transaction model aims at supporting collaborative work [Anf00b].

As mentioned in chapter 3 there is a broad consensus in the database research community that classical concurrency control is too restrictive to support a number of application domains. Apotram increases concurrency by introducing two new correctness criteria respectively called *conditional conflict serializability (CCSR)* and *nested conflict serializability (NCSR)*. These correctness criteria are generalizations of classic conflict serializability (CSR, see section 2.4), which makes Apotram able to support CSR as a special case.

The formal CCSR and NCSR correctness criteria are realized and made practical by introducing two corresponding mechanisms for controlling concurrency control: *parameterized access modes* for CCSR and *nested databases* for NCSR. Parameterized access modes, nested databases and their corresponding criteria will be presented in section 4.2 and 4.3, respectively.

The two correctness criteria can be combined to form the *nested conditional conflict serializability (NCCSR)* correctness criterion, which will be discussed in section 4.4. Transactions that follow the NCCSR criterion are said to have ACCID (pronounced *axid*) properties: *Atomicity, Consistency, Conditional Isolation, and Durability*. CCSR makes the isolation conditional by letting the application conditionally allow conflicts.

The content of this chapter is largely based on [Anf97].

4.2 Parameterized Access Modes

Traditional serializability considers two operations conflicting if at least one of them is a write. Thus, read-write and write-read are considered conflicting. Only read-read actions are considered non-conflicting. This compatibility relationship is illustrated by figure 4.1.

	R	W
R	*	
W		

Figure 4.1: The traditional compatibility matrix. The asterisk indicates compatibility. Only concurrent reading is considered non-conflicting.

However, this definition of access compatibility is too restrictive for many application domains. The idea of parameterized access modes is to *conditionally* allow read-write and write-read conflicts. Write-write conflicts are still considered conflicting and will be looked into in section about nested databases. The formal correctness criterion, *Conditional Conflict Serializability (CCSR)*, used by parameterized access modes will be presented in the next section.

The condition for determining if two operations conflict is given by *parameterized access modes*. Parameterized access modes give the application the possibility to associate its read and write accesses with parameter values. The parameters values are used by to indicate whether read-write or write-read accesses should conflict. Parameterized read and write will be represented by R(A) and W(A), respectively, where A is the parameter value. Parameters are subsets of some application-defined parameter domain. Two accesses, R(A) and W(B) are defined as compatible iff $B \subseteq A$. The resulting compatibility matrix is depicted in figure 4.2.

	R(A)	W(B)
R(A)	*	?
W(B)	?	

Figure 4.2: The CCSR compatibility matrix. Asterisk indicates compatibility, question mark indicates compatibility iff $B \subseteq A$. The read-write and write-read conflicts are made conditional.

The following example taken from [Anf00a] makes this concept more concrete:

> ... assume the set of available parameter values is good, medium, bad. Then e.g. W(bad) and R(good) would conflict, while

W(good) and R(medium, good) would not. The idea is that a writer using W(good) signals to any potential reader that the reliability or quality of the W-locked data is "good". Conversely, a reader using R(medium, good) thereby tells the system that he/she is willing to read data that is of "good or medium" quality/reliability.

Non-parameterized read and write modes can be represented as R and W, but they will be treated as R($\emptyset$) and W($*$), respectively, where * denotes some arbitrary superset of D. D is the domain of all parameter sets, thus $*$ will be a superset of all allowed parameter sets. When using the non-parameterized access modes, the condition of the conditional conflict will never be fulfilled and the behavior falls back to classic conflict serializability. This shows that CSR is a special case of CCSR.

Figure 4.3 gives some examples of parameterized access.

Access Mode	Access Mode	Relation	Compatible
R	W	$\{*\} \not\subseteq \{\emptyset\}$	No
R$\{a\}$	W$\{a\}$	$\{a\} \subseteq \{a\}$	Yes
R$\{a,b\}$	W$\{a\}$	$\{a\} \subseteq \{a,b\}$	Yes
W$\{a\}$	R$\{a,b\}$	$\{a\} \subseteq \{a,b\}$	Yes
R$\{a,b\}$	W$\{a,c\}$	$\{a,c\} \not\subseteq \{a,b\}$	No
R$\{\emptyset\}$	W$\{a\}$	$\{a\} \not\subseteq \{\emptyset\}$	No

Figure 4.3: Examples of parameterized accesses. The first and second columns show the competing access modes. The third column shows the conditional conflict relation. '$*$' denotes some arbitrary superset of the domain of parameters. Notice that *all* write access modes conflict with R$\{\emptyset\}$.

4.2.1 Conditional Conflict Serializability (CCSR)

The formal correctness criterion resulting from making conflicts conditional is called *Conditional Conflict Serializability (CCSR).*

The motivation behind CCSR is twofold:

1. Enable customization of the notion of a conflict.

2. Enable communication of the quality of uncommitted data.

When using parameterized access modes these goals are achieved by using appropriate parameters. The formal definition of CCSR results when substituting the notion of a conflict in traditional serializability theory (see section 2.4, page 23) with the conditional one given in the following definition:

Definition 5 (Conditional Conflict) *The parameterized read mode R(A) and the parameterized write mode W(B) conflict unless* $B \subseteq A$.

Recall the definition of conflict equivalence given in definition 1 on page 23. By using the conditional notion of a conflict the definition of *conditional conflict equivalence of histories* results.

If we define serializability by using this definition of history equivalence, we get conditional conflict serializability:

Definition 6 (Conditional Conflict Serializability (CCSR)) *A history is* conditional conflict serializable *iff it is conditional conflict equivalent to a serial history.*

The serializability theorem (page 26) states that a history is serializable iff the *serialization graph (SG)* is acyclic. The CSR corresponding serialization graph, *conditional conflict serialization graph (CCSG)*, can be defined if conflicts are understood in the conditional sense. Then, by replacing SG with CCSG in the theorem, we get the generalized Serializability theorem:

Theorem 3 (The Generalized Serializability Theorem) *A history H is serializable iff CCSG(H) is acyclic.*

The proof for this theorem could basically be a verbatim copy of the analogous proof of Bernstein et al [BHG87, page 33], provided SR is replaced with CCSR, SG(H) is replaced with CCSG(H), and conflict is understood in the conditional sense. It can be found in [Anf97, page 30].

4.2.2 CCSR and Correctness

In section 2.3.1 on page 20 we looked at some problems introduced by concurrency. What consequences does the relaxation of serializability allowed by CCSR have regarding these problems?

The Lost Update Problem. Both CSR and CCSR prevent write-write conflicts, therefore this problem is avoided by both criteria.

Dirty Read Problem When dirty reads are allowed under CSR, transactions have no way of knowing whether the data it reads is reliable or not. CCSR, on the other hand, always tells its transactions the quality/reliability of the data through parameter values. Instead of allowing all dirty reads, a transaction running under CCSR can explicitly tell what category of unreliable data it is willing to read by setting corresponding read parameters. Data items which are accessed with other write parameters will not be visible in the scope of the reading transaction. Thus, CCSR offers a more favorable solution to the dirty read problem than CSR.

The Incorrect Summary problem Given the fact that CSR is a special case of CCSR, and that this problem is avoided in CSR, CCSR can also avoid this problem by using ordinary read modes. However, the problem can be made arbitrarily small by proper use of access modes or protocols as discussed in [Anf97, pages 35–37]

The Unrepeatable Read Problem Insisting on repeatable reads in environments (e.g. concurrent engineering environments) where data is constantly being altered seems somewhat unreasonable. The demand for repeatable reads could on the other hand be significant in environments where complex queries coexist with short update transactions [Anf97].

The unrepeatable read problem can be eliminated by introducing "enhanced" read parameters that dynamically establish integrity constraints. The reader is referred to [Anf97, pages 37–38] for an in-depth discussion.

As already discussed the relaxation of serializability is necessary to support demands made by a number of application domains. The price to pay for giving the applications more flexibility is the reoccurrence of some of the problems eliminated by traditional conflict serializability. It is therefore important to investigate when these classes of concurrency problems result in actual problems for the specific application.

4.2.3 Relationship of CCSR to the Traditional Transaction Classes

Recall that CSR is a special case of CCSR. This means that the class of CSR histories must be a proper subset of the class of CCSR histories. Figure 4.4 shows the how CCSR relates to the traditional classes of transaction histories. Notice that, unlike CSR, CCSR allows histories that are *not* view serializable (VSR). For the proofs of these relationships the reader is directed to [Anf97, page 38].

4.2.4 Enforcing CCSR

The following discussion assumes that locking is used as the concurrency control scheme (locking was discussed in section 2.5). Parameterized access modes can be realized by associating each lock with the parameter of the access mode. The conflict detection algorithm of the lock manager will now have to use the *conditional* notion of a conflict. When scheduling a lock request for a resource x with a given access mode, the lock manager has to test that the requested parameterized lock is compatible with all the held locks on x. Figure 4.5 gives an example of a history of parameterized lock requests on a resource. Request 3 is a non-parameterized write-lock request

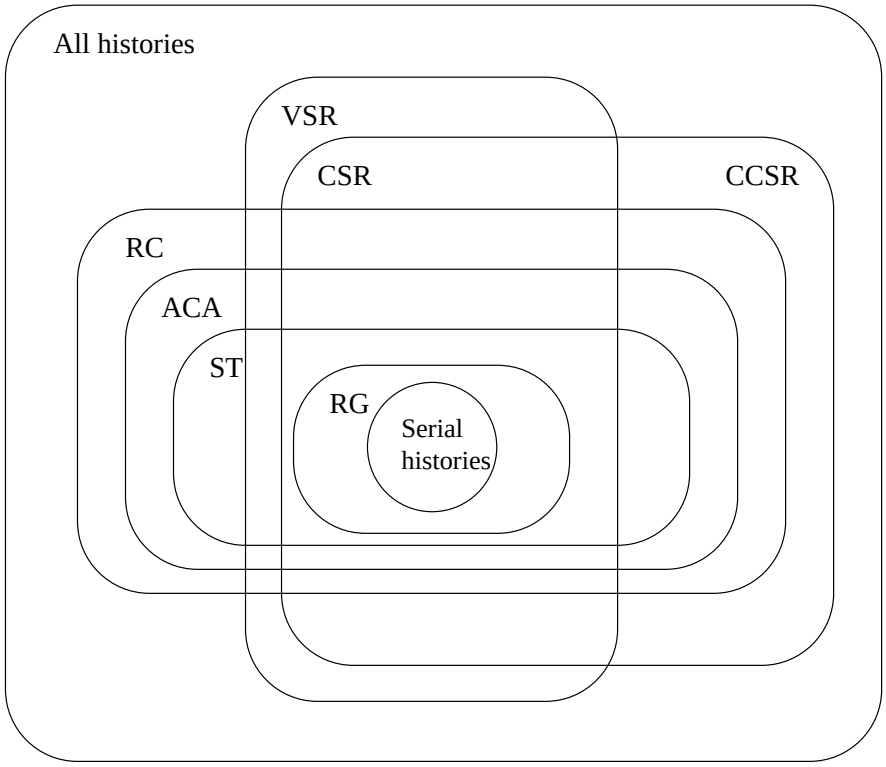


Figure 4.4: Relationship of CCSR to traditional transaction classes. No rectangle for CSR; it is the intersection of CCSR and VSR. This figure is adopted from [Anf97].

that is denied because of the existing read-locks. Request 4 is denied because request 2 stated that it is only willing to read from transactions which write with the b parameter ($\{a\} \not\subseteq \{b\}$). Request 6 conflicts with 5, and request 7 is denied because write-write accesses always conflict.

Just as 2PL ensures conflict serializability (see page 29), parameterized 2PL will ensure conditional conflict serializability (CCSR). Parameterized 2PL uses parameterized locks and the conditional notion of a conflict described above.

The size of each lock is an important performance metric in transaction managers. The number of locks can be significantly large, which makes it important that each lock requires as little memory as possible to avoid a heavy memory footprint. The implementation therefore has to make sure that the additional memory requirement induced by lock parameters is small. The set inclusion tests must also be efficient to avoid significantly reducing performance when scheduling lock requests.

No.	Lock Request	Lock Granted	r locks	w locks
1	$rl\{a, b\}$	Granted	rl:None	wl:None
2	$rl\{b\}$	Granted	rl:$\{a, b\}$	wl:None
3	wl	Denied $(1, 2)$	rl:$\{a, b\}, \{b\}$	wl:None
4	$wl\{a\}$	Denied (2)	rl:$\{a, b\}, \{b\}$	wl:None
5	$wl\{b\}$	Granted	rl:$\{a, b\}, \{b\}$	wl:None
6	$rl\{c\}$	Denied (5)	rl:$\{a, b\}, \{b\}$	wl:$\{b\}$
7	$wl\{b\}$	Denied (5)	rl:$\{a, b\}, \{b\}$	wl:$\{b\}$

Figure 4.5: Examples of parameterized lock requests. All requests are for locks on the same resource. Time grows downward. The 'Lock Granted' column shows if the request was granted or denied. If it was denied the numbers in parentheses indicate which previous lock requests it conflicted with. The last two columns show the existing read and write locks on the resource prior to the request. The w lock request in line 3 represents a non-parameterized lock request.

4.2.5 Example Scenario

This example is from the domain of software engineering applications. Imagine a team of engineers working on a set of source files. To be able to access a file the engineers must lock it in the appropriate way. During the period an engineer, say Bill, is doing his changes to his locked source files, other engineers might want to look at Bill's work. When using classic serializability, isolation prevents other engineers to see Bill's intermediate results. As long as engineers are aware of which files are in development and of their level of reliability, there really is not any need to prevent browsing. Using parameterized access modes makes it possible to relax isolation in a controlled manner. The parameter values could indicate the reliability of data items, e.g. *low*, *medium*, and *high*. At early stages Bill could use W(*low*) locks to indicate low reliability and upgrade the parameter as his files become more reliable. Other engineers that want to browse indicate what level of reliability they are willing to accept by setting corresponding read parameters (e.g. R(*medium*)).

4.3 Nested Databases

While parameterized access modes allow relaxation of serializability by conditionally allowing read-write and write-read conflicts, nested databases deal with write-write conflicts and enable controlled collaborative work between two or more transactions.

The nested database concept is inspired by Spheres of Control described on page 42. A database can be thought of as having a Sphere of Control

including the data items of the database. This Sphere of Control will be called a DBSOC. Transactions running within the database create their own Spheres of Control consisting of the data items they access. Based on what access mode is used, two types of spheres of control emerge: *Read SOCs (RSOCs)* and *Write SOCs (WSOCs)*. Allowing transactions to dynamically convert their WSOCs into DBSOCs gives rise to the idea of nested databases. The database that results from this dynamically created DBSOC is called a *subdatabase*. Other transactions can execute within this dynamically created DBSOC on behalf of its owner.

The creating transaction is said to be the *owner* of the subdatabase. A subset of the owner's write-locked data items (WSOC) can be logically moved into the subdatabase. All transactions must run in the context of one and only one database. The transaction is said to *visit* that database and may only write-lock and modify objects that are contained by the database it visits. Top level transactions are said to visit the global database. When an owner transaction moves a data item into its subdatabase, the data item is removed from the database it is moved from. This way nested databases form a hierarchy of disjoint logical sets of data items. It is important to emphasize that the term *'database'* is used in a logical and not physical sense. When stating that objects are contained by or moved into a database, it does not mean that the items are physically contained or moved.

A transaction that visits a subdatabase may recursively create its own subdatabase. This allows recursion to arbitrary depths or to a possibly maximum depth chosen by the implementation.

The owner of a subdatabase may choose to *commit* the subdatabase when no transactions are visiting it. If visiting transactions exist, the owner could have the choice of forcing them to commit or abort. The objects of the subdatabase are then moved back into the WSOC of the owner. The owner may also choose to *abort* the subdatabase, which results in all work done in the context of the subdatabase, and in all its inferiors, being rolled back. This is comparable to the abortion of a parent transaction in the nested transaction model described in section 3.8, page 44.

The owner of a subdatabase may be able to dynamically enforce access control over its database. It can for example define a user set that specifies the users that are authorized to visit the database and with what access modes (e.g. only allow browsing).

Work done by visiting transactions commit their results to the owner of the database, which can choose to accept, reject(rollback) the changes or refuse the commit request (e.g. if the owner regards the work to be committed as unfinished).

Figure 4.6 gives a summary of the above described rules of nested databases and figure 4.7 illustrates the structure of transactions and nested databases.

Nested Database Rules:

- ⇒ A transaction is created in the context of a single database
- ⇒ A transaction can only access objects of the database it visits.
- ⇒ A transaction can dynamically create subdatabases
- ⇒ An owner of a subdatabase can move a selection of its write-locked objects into the subdatabase
- ⇒ An owner of a subdatabase can can commit or abort the subdatabase
- ⇒ An owner of a subdatabase can accept or reject work performed by visiting transactions.
- ⇒ An owner of a subdatabase can enforce an authorization policy on the subdatabase.

Figure 4.6: Nested database rules

4.3.1 Nested Conflict Serializability (NCSR)

Traditional conflict serialization is enforced on the level of the database. Nested databases gives the transactions the possibility to create their own databases. Remember that the nested databases form a *disjunct* hierarchy of sets of data items. The fact that these databases are disjunct results in concurrency control (e.g. CSR) having only to be enforced on a per database basis.

This recursive enforcement of CSR as correctness criterion at all levels of databases is referred to as *Nested Conflict Serializability (NCSR)*:

Definition 7 (Nested Conflict Serializability (CCSR)) *A history is* nested conflict serializable *iff all the following conditions hold:*

1. *Subdatabases can be nested to arbitrary depths.*
2. *Transaction histories in subdatabases are CSR.*
3. *Transactions in subdatabase histories commit to the subdatabase owner.*

4.3.2 Implementation of Nested Databases

Nested databases can be simply implemented by introducing a new lock mode called *DB lock*. The DB-lock has exactly the same compatibility relationship to other locks as the write-lock has. When a transaction creates a

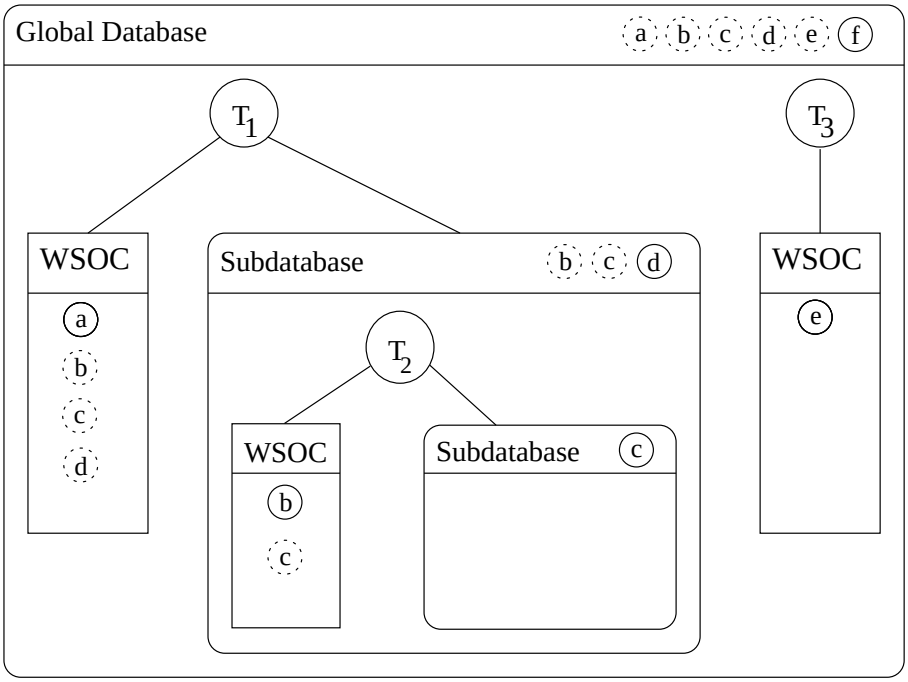


Figure 4.7: Nested databases. a, b, c, d, e, and f represent the data items made available by the global database. If the circle is dotted it means that the item has been moved to a lower level. The T_1 transaction has created a subdatabase containing the b, c, and d items. T_2 is a *visiting* transaction of the subdatabase and has created its own subdatabase containing c.

subdatabase consisting of some subset of its write-locked items, it *converts* its write-locks on the items to a DB-lock. To access the items inside the database, transactions must aquire ordinary locks on the items. The conversion from W to DB will never have to be delayed or rejected, but converting from DB back to W can only be allowed if no other locks are being held in the domain of the DB-lock.

4.3.3 Enforcing NCSR

The two-phase locking protocol, which ensures CSR transaction histories, was introduced in section 2.5.2 on page 27. In [Anf97], the author shows that database nesting will result in NCSR histories, provided two-phase locking is employed in all subdatabases.

4.3.4 Example Scenario

Let us return to the domain of software engineering. Imagine a software team consisting of Linda, Bill, and Janet. Linda is the team lead and has the responsibility of extending a software module with some functionality. The module consists of a number of source files. At the initiation phase of the project Linda creates a subdatabase consisting of the module's source files and gives Bill and Janet access privileges. Bill and Janet can now perform their work within the subdatabase. Each time Bill or Janet commit their work, it is committed to owner of the subdatabase, i.e. Linda. Linda can choose to accept,reject, or postpone the commit. Linda is also able to control requests to roll-back work.

Let us say that a part of the module has to be integrated with another module developed by a different team and that Linda has given Bill the responsibility of the integration. Bill has to collaborate with the other team and administrate the work being done. Bill can now create his own subdatabase, which will be nested within Linda's subdatabase, and give access to the members of the other team with whom he has to collaborate. Bill has full control over the work being done in his subdatabase and can commit the subdatabase when he is confident that the job is done. Bill then commits his transaction, which fate lies in the hands of Linda.

Notice that this way of working with nested databases poses a solution to the unpredictable developments problem on page 39.

4.4 Integrating NCSR and CCSR: NCCSR

Recall that NCSR deals with write-write conflicts and requires that the transactions histories in subdatabases are CSR. If we allow the histories of subdatabases to be of the more general correctness criterion CCSR, we generalize NCSR to *nested conditional conflict serializability (NCCSR)*:

Definition 8 (Nested Conditional Conflict Serializable (NCCSR)) *a transaction history is nested conditional conflict serializable iff:*

1. *Subdatabases can be nested to arbitrary depths.*
2. *Transaction histories in subdatabases are CCSR.*
3. *Transactions in subdatabase histories commit to the subdatabase owner.*

Note that CSR, CCSR, and NCSR are all special cases of NCCSR.

This brings the increased flexibility introduced by parameterized access modes into the domain of nested databases, thus making transactions able to deal with combinations of read-write,write-read, and write-write conflicts.

In section 4.3 we saw that nested databases are created by converting a WSOC (set of write-locks) to a DBSOC. If the write-locks are parameterized

then the resulting subdatabase will also be parameterized. The combination of parameterized access modes and nested databases lead to some issues that will be further looked into by the next chapters of this thesis.

4.5 Summary

This chapter has described the Apotram transaction model and its introduction of parameterized access modes and nested databases to support collaborative work.

At the time of writing, there exist two implementations of Apotram. The first is the result of a collaboration involving Sun Microsystems Laboratories (California, US), University of Glasgow (Scotland) and Apotram AS (Norway) [DA00a, DA00b]. This is a prototype implementation. In addition to the transaction model, a proof of concept application was implemented that exploits Apotram's collaborative features. The demo application simulates a multi-user word processing system. Both the Apotram prototype and the demo application were developed using the Java programming language [JSGB00]. The second implementation of Apotram is part of a development contract between EPM Technology and Apotram AS. It is implemented in the C programming language.

A third implementation aimed at the commericial market is currently being developed using Java and the Oracle DBMS.

Chapter 5

Dynamic Modification of Isolation

5.1 Introduction

In the previous chapter the Apotram transaction model was presented with its two new correctness criteria, conditional conflict serializability (CCSR), nested conflict serializability (NCSR) and their integration nested conditional conflict serializability (NCCSR). In [Anf97, page 55], Anfindsen states that Apotram should support dynamic modification of parameters. Thus, allowing transactions to dynamically change their applied parameters during their lifetime and thereby changing the degree of isolation. This results in some issues that have to be further investigated.

In this chapter we describe and analyze dynamic modification of parameters, its resulting issues, and purpose solutions. We first take a look at dynamic modification of parameters in the context of CCSR only. Then, we look at resulting issues when integrating NCSR.

5.2 Dynamic Modification of Isolation

Section 4.2 of chapter 4 discussed parameterized access modes. Recall that two parameterized accesses $R(A)$ and $R(B)$ are compatible iff $B \subseteq A$. The parameterized compatibility matrix is reprinted in figure 5.1 for convenience. This way parameterized access modes are used to conditionally relax isolation.

During the lifetime of a transaction it could be desirable to change the read parameters, write parameters, or both for some or all of its items. A transaction is created with a set of parameters defined by the application developer or user. The transaction should be able to change its parameters by notifying/invoking the transaction manager. The transaction manager

	R(A)	W(B)
R(A)	*	?
W(B)	?	

Figure 5.1: The CCSR compatibility matrix. Asterisk indicates compatibility, question mark indicates compatibility iff $B \subseteq A$. The read-write and write-read conflicts are made conditional.

has to determine the consequences of the parameter change and grant or revoke the modification.

Example. Let us return to the example of section 4.2.5 on page 57. In this example parameters are used to indicate the reliability of source files, e.g. *low*, *medium*, and *high*. Imagine that Bill is working on a new source file. He starts with W(*low*) locks to indicate low reliability. As his work reaches medium reliability, he modifies the write parameters of the source file to W(*medium*). He has now dynamically modified his write parameters from *low* to *medium*. This change has consequences on isolation. Imagine, that during Bill's period of work, Linda is using read parameter R(*medium*) to indicate that she is only interested in source files of medium reliability. At first, she will be denied to browse Bill's source file because he uses W(*low*), but after his parameter change she will be able to browse the file she previously did not have access to. Here, the parameter modification resulted in another person gaining access to a source file. But, imagine now that after Linda locks Bill's source file for browsing, Bill for some reason has to downgrade the reliability of the file back to W(*low*). If the transaction manager allows this parameter modification, it would violate the CCSR correctness criterion. Linda would be using R(*medium*) and Bill W(*low*) where *medium* $\not\subseteq$ *low*. The transaction manager must implement some policy to deal with the issues resulting from dynamic parameter modification. The goal of this chapter is to analyze the issues related to dynamic parameter modification in context of CCSR and NCCSR and suggest various policies to deal with them.

5.3 Scope of Parameters

[Anf97, page 55] argues that an Apotram transaction model should allow multiple concurrency levels within a transaction. This means that a single transaction should be able to use various access mode parameters. What scope should a transaction's parameter usage cover? It is assumed that locking is used for ensuring concurrency control and that parameterized access modes are implemented by associating each lock with a corresponding parameter value. Thus, each lock acquired by a transaction may use different

parameter values. This results in the scope being the data item. A better alternative would be to let the transaction be the scope of parameter operations. All data manipulation statements of a transaction are then executed using the parameter values associated with the transaction. If the transaction needs to execute some statements with other parameter values, it spawns a subtransaction that uses the desired set of parameters. This way there is no need to specify read/write parameters for each read/write lock. The multiple concurrency accesses are conveniently grouped by the structure of nested transactions.

The usage of different parameter sets in subtransactions leads to an issue that has to be investigated. Remember that under the classic nested transaction model (described in section 3.8, page 44) when a subtransaction commits, its locks are upward inherited by its parent. Now, imagine that a transaction, T, has spawned a subtransaction using a different parameter set. The subtransaction locks various items using its parameters. When the subtransaction has performed its work, it commits. As a result of the commit T inherits the locks. But, now T owns locks with different parameter values: the locks it has acquired using its parameters and the locks with the parameters inherited from the committed subtransaction. This violates the requirement that each transaction executes using only its associated parameters. This problem is illustrated in figure 5.2.

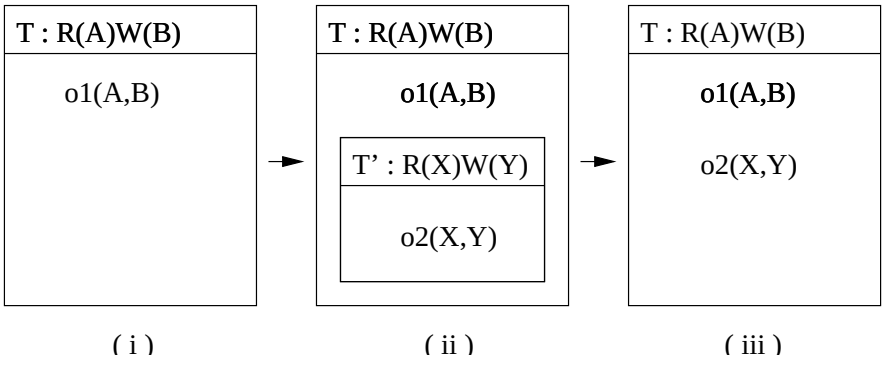


Figure 5.2: Transaction parameter scope problem. (i) The transaction T is using A and B for its read and write parameters, respectively. It is working on a data item, o_1. $o_1(A, B)$ means that the transaction has locked the o_1 data item with A for read parameters and B for write parameters. (ii) T needs to perform some work using different parameters, $R(X)W(Y)$. This is accomplished by spawning the T' subtransaction. T' performs its work on o_2 and commits. (iii) T inherits o_2 from T'. This results in a conflict because T is now operating with two different access modes: $R(A)W(B)$ and $R(X)W(Y)$.

One way to deal with this problem would be to convert the parameters of the subtransaction's locks to those of the parent when committing subtransactions. We will later see that this parameter conversion/modification will under some circumstances lead to conflicts and might, depending on the policy of the transaction manager, be denied.

5.4 Dynamic Modification of Parameters in CCSR

We will in this section concentrate on the consequences of dynamic modification of parameters in the context of conditional conflict serializability. Recall that CCSR is realized by parameterized access modes. We begin by analyzing the problem of parameter modification of read and write parameters separately. In the next section, 5.4.3, strategies for dealing with issues of parameter modification are discussed. In section 5.5 we increase complexity by integrating NCSR.

5.4.1 Modification of Read Parameters

Read parameters are used by a transaction to specify what type of dirty data it is willing to read. A transaction using A for its read parameters can read data written by any transaction that uses a subset of A for its write parameters. Note that concurrent read accesses of any parameter values *never* conflict. Thus, modification of read parameters will never introduce conflicts with other transactions that read common data regardless of the parameter values used. The modification can therefore only introduce conflict with writers.

Let us take a look at a sample history that includes modification of read parameters.

No.	T_1	T_2	Comment
1	$R\{a,b\}W\{a\}$	$R\{a\}W\{a,b\}$	
2	$rl(o_1)$		
3	$wl(o_2)$		
4		$wl(o_1)$	Ok: $\{a,b\} = \{a,b\}$
5		$rl(o_2)$	Ok: $\{a\} = \{a\}$
6	$R\{a,b\} \rightarrow R\{a,b,c\}$		Ok: $\{a,b\} \subset \{a,b,c\}$
7	$R\{a,b,c\} \rightarrow R\{b\}$		Not Ok: $\{a,b\} \not\subset \{b\}$

Here we see a history including two transactions. Step 1 shows the read and write parameters of each transaction. By looking at the read parameters we see that T_1 is willing to read a more extensive set of dirty data than T_2 because T_1's read parameter set is larger than T_2's. Both transactions are able to read data from transactions that use $W\{a\}$, but T_1 is additionally able to read from transactions that use $W\{b\}$ and $W\{a,b\}$. In the locking covered by steps 2 through 5, a dependency between T_1 and T_2 is established

because they read data items from each other. T_1 reads o_1 from T_2 and T_2 reads o_2 from T_1. In the comment column we see why this is allowed by the CCSR correctness criterion.

In step 6, T_1 modifies its read access mode from $R\{a,b\}$ to $R\{a,b,c\}$. This is an expansion of the read parameter set, which results in T_1 now being able to read the same dirty data as before and in addition data written by transaction using $W\{c\}$, $W\{a,c\}$, $W\{b,c\}$, and $W\{a,b,c\}$. If we look at the condition of CCSR stating that that a transaction using R(A) is able to read from any transaction(s) using $W(B)$ if $B \subseteq A$, we see that no expansion of the read parameter set will ever introduce a conflict.

Theorem 4 (Expansion of read parameters) *If the read parameters of a transaction are expanded, then no conflict will result.*

The proof comes straightforward given the following lemma:

Lemma 1 () $A \subseteq B \wedge B \subseteq C \Rightarrow A \subseteq C$
The proof follows easily from the definition of the subset relation and can be found in [Gri94, page 147].

If we substitute the set variables in the above lemma with our read and write parameter sets we get:

$$B \subseteq A \wedge A \subseteq A' \Rightarrow B \subseteq A'$$

This proves that the expansion of the write parameter set will never violate the CCSR condition.

In step 7 T_1 modifies its read access mode from $R\{a,b,c\}$ to $R\{b\}$. This modification introduces a conflict with T_2. T_1 is reading o_1 from T_2 and T_2 is using $\{a,b\}$ for its write parameters. After changing its read parameters to $\{b\}$, T_1 is no longer allowed to read from T_2 because it would violate the CCSR correctness criterion: $\{a,b\} \not\subset \{b\}$. Notice that the parameter modification of T_1 is a restriction of the read parameter set. Informally, T_1 is saying that it is no longer willing to read dirty data that is written with a or c write parameters. The conflict occurs because T_1 is already reading data written with conflicting write parameters. If T_2 would have used $\{b\}$ for its write parameters the read parameter set restriction would not have been a problem.

Thus, every time an element is removed from the read parameter set of a transaction, a conflict might be introduced.

To be able to formally specify and test conflicts some terminology and definitions are introduced in figure 5.3.

Given this terminology we can now define functions which will help us deal with parameter modifications.

Definition 9 $ReadConflictingTxns(T,A) = \{T_k \mid T_k \in activeTransactions \wedge T_k \neq T \wedge (T\ readsFrom\ T_k) \wedge wp(T_k) \not\subseteq A\}$

Terminology:

$activeTransactions$ is the set of all active transactions.

$rp(T)$ is a function that returns the read parameter set used by the transaction given as argument. E.g. $rp(T_2) = \{a\}$.

$wp(T)$ is a function that returns the write parameter set used by the transaction given as argument. E.g. $wp(T_2) = \{a, b\}$.

$rl(o)$ is a boolean function that is evaluated as true iff o is read-locked by one or more transactions. E.g. $rl(o_1) = true$.

$wl(o)$ is a boolean function that is evaluated as true iff o is write-locked by a transaction. E.g. $wl(o_1) = true$.

$readers(o)$ is a function that is returns the set of transactions that have read-locked o. E.g. $readers(o_1) = \{T_1\}$.

$writer(o)$ is a function that is returns the transaction that has write-locked o. E.g. $writer(o_1) = T_2$.

$rls(T)$ is a function that returns the set of data items that are read-locked by T. E.g. $rls(T_1) = \{o_1\}$.

$wls(T)$ is a function that returns the set of data items that are write-locked by T. E.g. $wls(T_1) = \{o_2\}$.

$readsFrom(T_1, T_2) = rls(T_1) \cap wls(T_2) \neq \emptyset$. This is a boolean function that returns true iff T_1 reads from T_2. Can also be used in infix notation (e.g. T_1 $readsFrom$ T_2)

Figure 5.3: Transaction terminology

The $ReadConflictingTxns$ function returns the set of transactions that will conflict with T if its read parameters are set to A. The last clause demands that A and T_k's write parameters must conflict. Considering the sample history above, we get that $ReadConflictingTxns(T_1, \{b\}) = \{T_2\}$. Thus, modifying the read parameters of T_1 to $\{b\}$ introduces a conflict with T_2.

A read parameter modification of a transaction T to A does not introduce any conflicts iff $ReadConflictingTxns(T, A) = \emptyset$. Please note that the outcome depends completely on the state of the system at the time the test is applied. A read parameter change allowed in one instance of time might be disallowed in another due to changes to the set of active transactions or changes to their write parameters.

Another useful function is one that returns the set of shared data items

that are involved in the conflict:

Definition 10 $ReadConflictingItems(T, A) = \{o_k \mid o_k \in rls(T) \wedge wl(o_k) \wedge wp(writer(o_k)) \not\subseteq A\}$

The function says that the data item has to be read-locked by T, write-locked by some other transaction, and the parameters of the writer and T must conflict. As with the $ReadConflictingTxns$ function, the modification of the read parameters of a transaction T to A does not introduce conflicts iff $ReadConflictingItems(T, A) = \emptyset$.

5.4.2 Modification of Write Parameters

Now, let us switch our attention over to the modification of write parameters. A history of two transactions is given below.

No.	T_1	T_2	Comment
1	$R\{a,b\}W\{a\}$	$R\{a\}W\{a,b\}$	
2	$rl(o_1)$		
3	$wl(o_2)$		
4		$wl(o_1)$	Ok: $\{a,b\} = \{a,b\}$
5		$rl(o_2)$	Ok: $\{a\} = \{a\}$
6		$W\{a,b\} \rightarrow W\{a\}$	Ok: $\{a\} \subset \{a,b\}$
7		$W\{a\} \rightarrow W\{a,c\}$	Not Ok: $\{a,c\} \not\subset \{a,b\}$

The notation and first 5 steps were explained in section 5.4.1 and will not be repeated here. Please recall that write-write accesses always conflict and therefore changes to a transactions write parameters only affects other readers.

In step 6 T_2 successfully narrows its write parameter set from $\{a,b\}$ to $\{a\}$.

We will now show that restriction of write parameters never introduces conflicts.

Theorem 5 (Restriction of write parameters) *If the write parameters of a transaction are restricted, then no conflict will result.*

The theorem can be proven by using lemma 1 on page 67. If we substitute the set variables in the lemma with our read and write parameter sets and swap the first two clauses (allowed due to the commutative law of $\wedge$) we get:

$$B \subseteq A \wedge B' \subseteq B \Rightarrow B' \subseteq A$$

This proves that restriction of the write parameter set from B to B' will never result in a conflict.

In step 7 we see that the modification of the write parameters of T_2 conflicts with the read parameters of T_1. The addition of the c parameter

requires that any reading transactions must include c in their read parameter set. T_1 does not include c and is therefore not allowed to read the dirty data written by T_2. This shows that the addition of a write parameter, a, of a transaction T introduces conflicts if there exists at least one other transaction that has read-locked data items written by T without having the added parameter in its read parameter set. Please note that there can exist multiple readers of each data item write-locked by a transaction. This is different from the read parameter modification case, where there can exist at *most* one writer for each read-locked data item.

Definition 11 $WriteConflictingTxns(T, B) = \{T_k \mid T_k \in activeTransactions \wedge T_k \neq T \wedge (T_k\ readsFrom\ T) \wedge B \not\subseteq rp(T_k)\}$

The function returns the set of transactions that will conflict with T if its write parameters are set to B. The last clause demands that the B' and T_k's read parameters must conflict. Considering the sample history above we get that $WriteConflictingTxns(T_2, \{a, c\}) = \{T_1\}$. Thus, modifying the write parameters of T_2 to $\{a, c\}$ introduces a conflict with T_1.

A write parameter modification to B by a transaction T does not introduce any conflicts iff $WriteConflictingTxns(T, B) = \emptyset$. Again, the outcome depends on the state of the system at the time the test is applied.

The write equivalent version of $ReadConflictingItems$ is as follows:

Definition 12 $WriteConflictingItems(T, B) = \{o_k \mid o_k \in wls(T) \wedge \exists t_i[t_i \in readers(o_k) \wedge B \not\subseteq rp(t_i)]\}$

The function says that the object has to be write-locked by T, read-locked by at least one other transaction, and the parameters of at least one of the readers must conflict with B. As with the $WriteConflictingTxns$ function, the modification of the write parameters of a transaction T to B does not introduce conflicts iff $WriteConflictingItems(T, B) = \emptyset$.

5.4.3 Mechanisms for Dealing with Resulting Conflicts

To avoid introducing conflicts and thereby violating the correctness criterion, the transaction manager has to implement a strategy to deal with parameter changes. We will now present and discuss various strategies. The first strategy prevents conflicts from being introduced in the first place, while the rest provides mechanisms to deal with conflicts if they occur.

Queuing Parameter Modification Requests

The startegy is based on queuing requests. This technique is used for example in two phase locking (see page 27), which queues requests to avoid breaking conflict serializability. If a request for parameter modification results in one or more conflicts, then the transaction is suspended and the

request is queued. If at some later point in time the requested parameter modification is legal, then the modification is performed and the transaction regains control.

Apotram is designed to support environments where transactions can be long-lived (for long lived transactions see section 3.5). The above strategy would result in transactions having to wait indefinitely when modifying parameters. This is exactly what one wants to prevent in environments with long lived transactions, which makes this approach undesirable.

Only allow transactions to loosen isolation

This strategy only allows transactions to loosen isolation, never increase it. This is enforced by making parameter modification follow the rules from theorem 4 and 5:

1. Only allow expansion of the read parameter set:
   $R(A) \rightarrow R(A')\ iff\ A \subseteq A'$

2. Only allow restriction of the write parameter set:
   $W(A) \rightarrow W(A')\ iff\ A' \subseteq A$

Let us turn back to the example above where parameters were used to indicate data item reliability. Consider once more the work Bill does on a new source file. He starts with the *low* parameter and proceeds through *medium* to *high* as the reliability increases accordingly. Assume that the transaction manager enforces the above described policy. How should the parameters be set up in this example? It is natural to believe that more and more transactions are willing to read dirty data as the reliability of the data increases. This works well with rule 2 above, which states that the write parameter set can only be restricted. Thus, possibly giving more transactions access to the data.

In order to accomplish this with the Apotram parameters, the *low*, *medium*, and *high* categories can be mapped to parameter sets in the following way:

1. $high \rightarrow \{a\}$

2. $medium \rightarrow \{a, b\}$

3. $low \rightarrow \{a, b, c\}$

As one can see, the lower the level of reliability, the wider the parameter set. If Bill uses the write parameters $W(medium)$, and another user, Jane, read parameters $R(high)$, then the two accesses will conflict with each other because

$$W(medium) = \{a, b\} \not\subseteq \{a\} = R(high)$$

But, if Jane uses $R(low)$, the access is allowed:

$$W(medium) = \{a, b\} \subseteq \{a, b, c\} = R(low)$$

Rounding up, a transaction can only upgrade the level of reliability of the data it writes and read data of increasingly lower levels of reliability.

This approach puts significant restrictions on the modification of read and write parameters. On the other hand, an implementation of this approach would be straightforward and performance efficient. The only effort needed to support parameter modification would be a test that the new read parameter set is a superset of the current one and that the new write parameter set is a subset of the current one.

Only Allow modification of parameters if no conflicts result

The idea here is to allow parameter modification only if it does not introduce conflicts. Every time a transaction requests a parameter change the transaction manager must test if the change is allowed. The pseudo code for implementing a test to determine if a read parameter modification would introduce conflicts could be:

```
function testReadParameterModification
( T  : Transaction,
  A : ParameterSet ) : boolean
begin
   if rp(T) subset A then return true;

   for each dataitem o in rls(T) do
      if wl(o) and wp(writer(o)) conflicts with A then return false;
   next;

   return true;
end
```

Here, the transaction manager has to traverse the, possibly large, set of read-locked data items to determine if a read parameter modification should be allowed. The implementation has to make sure that the joined operation of testing and parameter modification is atomic. No transaction should during the test be able to change the state in a way that would introduce a conflict. Imagine that during an execution of this test the data item o_1 is cleared and the test continues to iterate through the rest of the read-locked objects. Imagine further that another transaction with conflicting write parameters chooses to write-lock o_1 during the remaining execution of the test and that the remaining test is successful. The test would return true even though the parameter change would result in a conflict. This problem

is analogous to the incorrect summary problem described in section 2.3.1, page 22.

The corresponding test for modifying write parameters is similar, with the exception that each write-locked data item can be read by multiple transactions, not just one. This would add an inner loop to the above $foreach$ loop, traversing the set of readers of the currently treated write-locked data item o. Thus, the write test is performance intensive because its growth rate is quadratic, $O(N^2)$[1], due to the nested for loop. The read test, on the other hand, has a linear, $O(N)$, growth rate.

Aborting competing transactions

This approach avoids potential conflicts by aborting the transactions that conflict with the new parameter set. Modifying read parameters aborts conflicting writers (at most one writer per data item), and modifying write parameters aborts conflicting readers (possibly multiple readers per data item). The following is a more formal description of this strategy:

- $T : R(A) \rightarrow R(A') \Rightarrow abort(ReadConflictingTxns(T, A'))$
- $T : W(B) \rightarrow W(B') \Rightarrow abort(WriteConflictingTxns(T, B'))$

This is a very aggressive approach in that the accumulated work of a possibly large set of transactions would be rolled-back because a transaction decides to modify its parameters.

Let us take a look at this strategy in the context of the example above. Kate is observing the work being done by Bill indicating *medium* reliability. Kate then decides that she is not willing to see work with less than *high* reliability and therefore modifies her read parameters to $R(high)$. This modification introduces a conflict with Bill's $W(medium)$ access mode and by applying the described approach all Bill's work will be aborted and rolled back. This is clearly not acceptable behavior in the context of this example.

Due to the consequences of this approach it is likely to believe that it would be useful in only a few special cases.

Release conflicting locks

Recall that a parameter modification results in a conflict if at least one data item is locked by another transaction with parameters that conflict with the requested parameters. The previous approach resolved the conflict by affecting the other transactions. This approach only affects the transaction that requests the parameter modification. It resolves conflicts by releasing all locks of the requesting transaction on data items that are involved in the conflict. Modifying read parameters leads to releasing conflicting read locks.

[1] This notation is called Big-O notation and states that the growth rate is quadratic.

When modifying write parameters, on the other hand, conflicting write locks have to be released.

- $T : R(A) \rightarrow R(A') \Rightarrow release(ReadConflictingItems(T, A'))$
- $T : W(B) \rightarrow W(B') \Rightarrow release(WriteConflictingItems(T, B'))$

A significant advantage of this approach is if the transactions follow the rule of two phase locking (see section 2.5.2), they will be forced into their shrinking phase when they release a lock. Thus, not being able to acquire new locks. As we saw in section 2.6 the time of read and write lock release has consequences for the properties of transaction histories generated. Let us take a look at the consequences for read and write locks separately:

Read. The premature release of read locks would prevent histories that are rigorous (a description of rigorous histories was given on page 33). Rigorous histories demand that read locks are held until commit. Strict histories could still be enforced because write locks are held till commit.

Write. The premature release of write locks prevents histories that avoid cascading aborts (ACA). When the write lock of a data item is released, the associated write parameter is removed. This makes transactions able to read the item without knowing that it is dirty and depends on the success of the transaction which had the write lock. If the transaction which released the write lock has to abort, then the reading transaction(s) would have to be rolled back too. Hence, this is not a desirable solution.

It has to be investigated if it is acceptable to sacrifice rigorous histories.

Split conflicting data items

Instead of releasing the locked items and loosing transactional control over them, one could use the split mechanism of *dynamic restructuring of transactions* (described in section 3.9) to split the locked items into a new transaction. When a read parameter change is requested that would introduce conflicts, the conflicting data items are split into a new transaction which uses the old parameters. Thus, a parameter modification can result in two transactions, the current transaction using the new parameters, a new transaction using the old parameters holding the items that would have introduced a conflict. Let us take a look at this strategy in the case when read parameters are modified:

1. T requests a read parameter change from A to A'.
2. Parameter change introduces conflicts: $ReadConflictItems(T, A') \neq \emptyset$.

3. T is split creating T' which uses the old parameters $R(A)W(B)$ and read-locks $ReadConflictItems(T, A')$.

4. T continues with the new parameters $R(A')W(B)$.

The process is illustrated by figure 5.4.

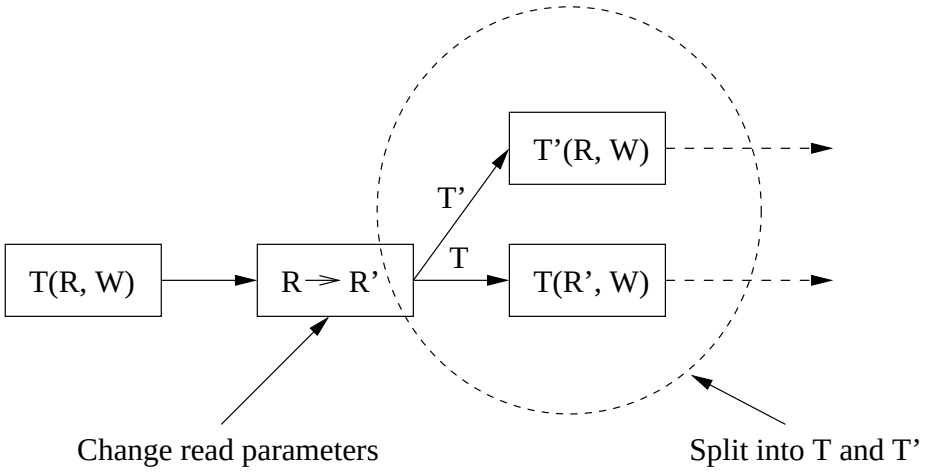


Figure 5.4: Modification of read parameters by splitting

The process for modifying write parameters:

1. T requests a write parameter change from B to B'.

2. Parameter change introduces conflicts: $WriteConflictItems(T, B') \neq \emptyset$.

3. T is split creating T' which uses the old parameters $R(A)W(B)$ and read-locks $WriteConflictItems(T, B')$.

4. T continues with the new parameters $R(A)W(B')$.

Figure 5.5 illustrates the modification of write parameters.

The split operations required when modifying read and write parameters respectively are:

Read parameter modification :

$Split - Transaction($
$T : (rls(T) \setminus ReadConflictingItems(T, A'), wls(T)),$
$T' : (ReadConflictingItems(T, A'), \emptyset));$

Write parameter modification: :

$Split - Transaction($
$T : (rls(T), wls(T) \setminus WriteConflictingItems(T, B')),$
$T' : (\emptyset, WriteConflictingItems(T, B')));$

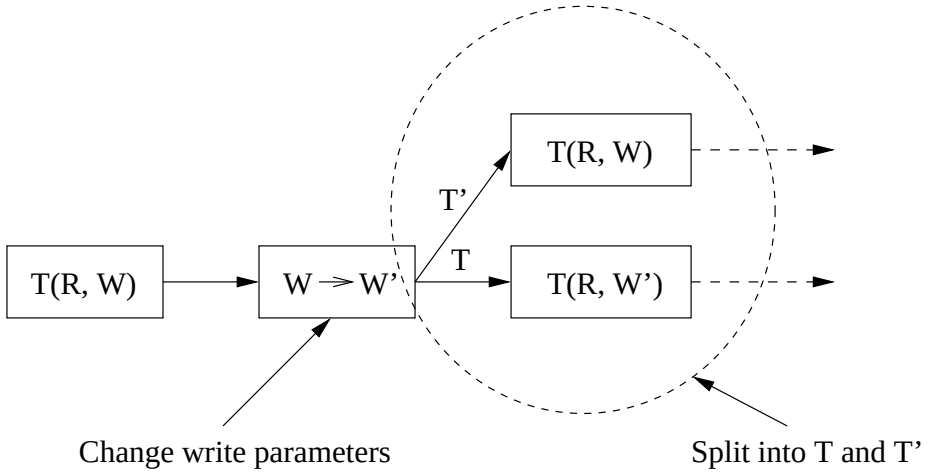


Figure 5.5: Modification of write parameters by splitting

Notice that the resulting transactions, T and T', are disjoint. They have disjoint read and write lock sets: $(rls(T) \cup wls(T)) \cap (rls(T') \cup wls(T')) = \emptyset$.

Split commit conflicting data items

This approach is identical to the previous one with the exception that the split transaction T' is committed immediately. See figures 5.6 and 5.7 for read and write parameter modification, respectively.

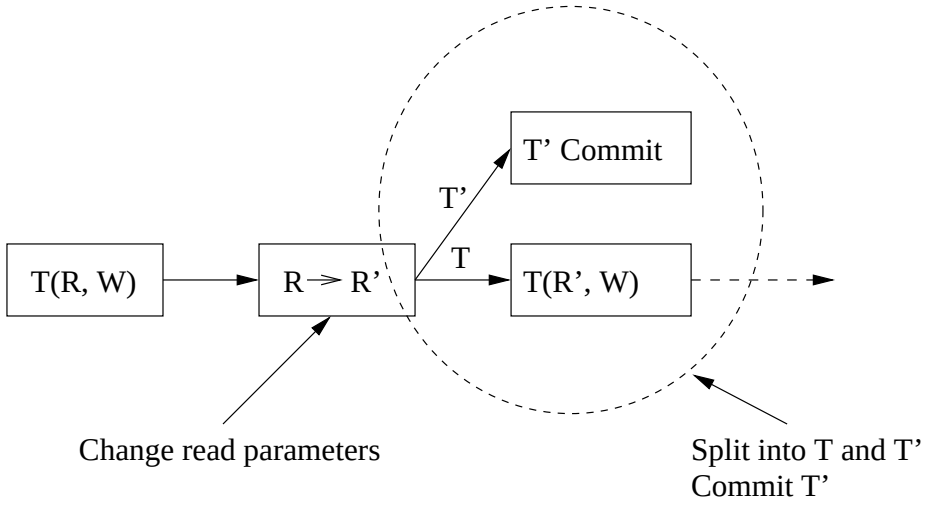


Figure 5.6: Modification of read parameters by splitting

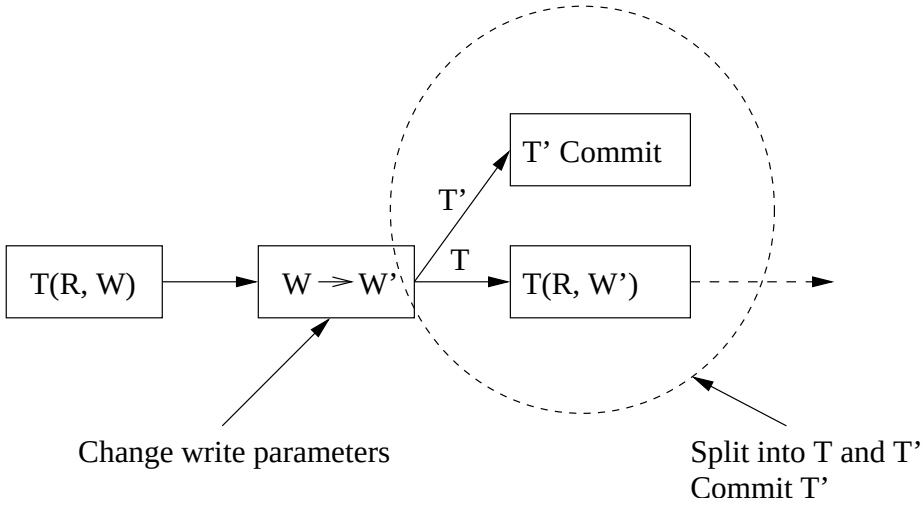


Figure 5.7: Modification of write parameters by splitting

Move conflicting read locks into subtransaction

In section 5.3 we looked at how multiple concurrency levels within a transaction was realized through subtransactions. We can use multiple concurrency levels to avoid the conflicts of parameter modification. The idea is to keep the old parameters for the data items that conflict with the new parameters. Thus, establishing two concurrency levels, one using the new parameters and the other retaining the old parameters. The pseudo code for a read parameter modification using this strategy could be:

```
// Create a new concurrency level
T' = CreateSubTxn(T, ReadConflictingItems(T, A'),A , B);

// Perform read parameter modification
T: A -> A'
```

But, as we saw in section 5.3 and figure 5.2 the commit or abort of the subtransaction,T', will result in the parent,T, inheriting the items locked by T'. These parameters of these items will have to be converted to those of T, which was precisely what had to be prevented to avoid conflicts. One way to deal with this is to prevent the commit of these subtransactions till no conflicts would result. Conflicts could be explicitly removed when they are caused by read parameters. The solution would be to append the subtransaction's read parameters to those of the parent. Now, the commit of the subtransaction would only result in the expansion of the read parameters of its locked items. One the other hand, this method requires the expansion of the parent's read parameters, something that might not be desirable. One

might at first think that this method for dealing with read parameters could be applied to the write case by only restricting the parameters instead of expanding them. However, this would result in an empty write parameter set($W(\emptyset)$) if the parent's and subtransaction's write parameters are disjoint. The $W(\emptyset)$ is not allowed because it would result in transactions requiring full isolation being able to read dirty data. Recall, that unparameterized transactions use $R(\emptyset)$ to avoid reading dirty data and $R(\emptyset)$ does *not* conflict with $W(\emptyset)$.

5.4.4 Summary

We have taken a look at dynamic parameter modification in the context of the CCSR correctness criterion. It was shown that increasing the degree of isolation results in the possibility of transactions loosing their granted right to observe data of the transactions with the now higher degree of isolation. The restriction of read parameters and expansion of write parameters of a transaction increases the transaction's degree of isolation. The expansion of read parameters and restriction of write parameters, on the other hand, relaxes isolation and does not introduce conflicts.

The following strategies for avoiding or dealing with introduced conflicts were presented and discussed:

1. Queuing Parameter Modification Requests
2. Only allow transactions to loosen isolation
3. Only allow modification of parameters if no conflicts result
4. Aborting competing transactions
5. Release conflicting locks
6. Split conflicting data items
7. Split commit conflicting data items
8. Move conflicting read locks into subtransaction

Strategies 1 and 2 prevent conflicts due to parameter modification, while 3 disallows parameter modifications that introduce conflicts. The rest of the strategies provide mechanisms to deal with conflicts introduced by parameter modification. Of these strategies 1 and 4 were found to be of limited applicability. It is important to notice that the mechanisms do not have to be applied to the transaction that requests the parameter modification. They could be applied to any transaction that is involved in the conflict. For example, say a transaction, T_1, decides to modify its write parameters and this introduces some conflict with the read parameters of another transaction, T_2.

Instead of dealing with the conflict in T_1, T_2 could be required to split the data items involved into a new transaction using read parameters which are compatible with the new write parameters of T_1. This could be used in the above example where Kate is browsing Bill's work only accepting *medium* reliability and Bill decides to degrade the reliability of his transaction to *low*. Instead of making this Bill's problem, Kate could be forced to split the data she is reading from Bill into a new transaction which accepts data of *low* reliability.

Please recall that when a transaction modifies its parameters, a significant number of transactions could be involved in a resulting conflict. By moving the responsibility of resolving the conflict away from the parameter modifying transaction, the conflict has to be dealt with in *all* the other transactions involved.

5.5 Integrating Parameterized Access Modes and Nested Databases (NCCSR)

The concept of nested databases was discussed in section 4.3, page 57. A transaction is always started in the context of one database and is said to *visit* that database. The transaction can begin (i.e. create) and commit or abort (i.e. end) nested databases. The transaction that created the nested database is called the *owner* of the database and can assign appropriate write parameters to it. The owner also controls what set of transactions/users that are allowed access to its nested database(s). Visiting transactions can recursively create their own nested databases resulting in a hierarchy of nested databases. A visiting transaction can only modify data items of the database it visits. However, it can read data items from any descendant database of its visiting database if the read access parameters used do not conflict with the write parameters of the nested database or any of its visiting transactions. Thus, data items of a nested database, *ndb*, can be read in two ways, by visiting transactions of *ndb* or by transactions visiting some superior database of *ndb*. Transactions that follow the latter approach will be called *observers* of a nested database. Please note that transactions visiting the global database can observe any data item if the access mode parameters do not conflict. The relation between visiting and observing transactions is illustrated in figure 5.8.

5.5.1 Nested Database Parameters

Recall that when a transaction moves a write-locked data item into a nested database, the write-lock of the item is converted into a nested database lock (NDB lock). NDB locks behave as ordinary write locks, so as the write-lock, the NDB lock can have associated parameters. Only the owner of a nested

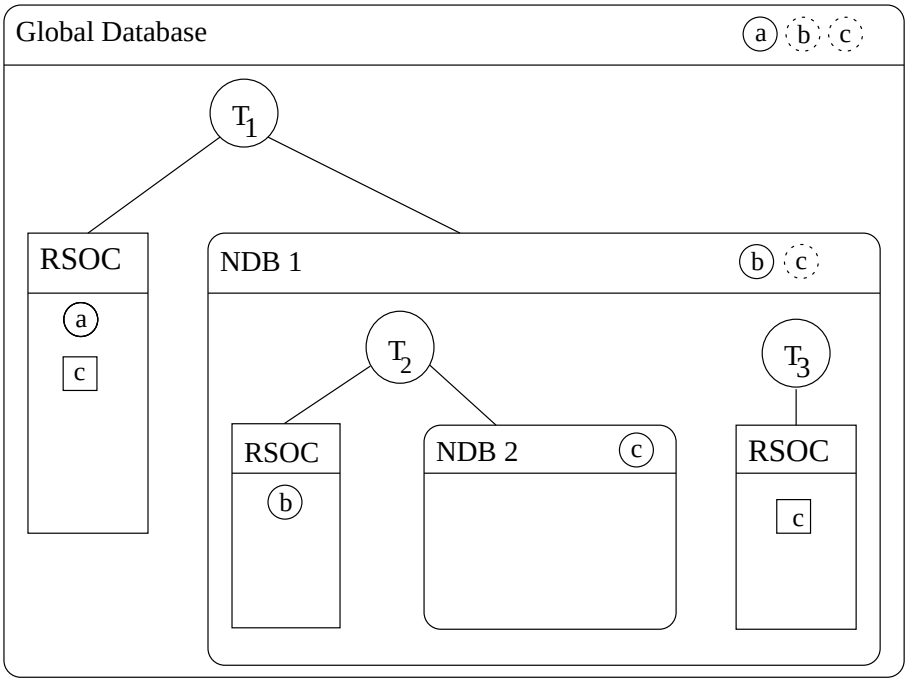


Figure 5.8: Visiting and observing read accesses of nested databases. The $WSOC$s of the transactions are not shown. Circled data items within $RSOC$s represent visiting read accesses and boxed data items represent observing read accesses. T_3 is an example of an observing transaction. It reads c from the descendant $NDB2$. T_1 both reads a from the database it visits and observes c from the descendant nested database $NDB2$. T_2 does not observe. It reads b from the database it visits and is the creator of $NDB2$.

database can move data-items into it. Remember that we have restricted transactions to use only one parameter value for all its locked data items, and that multiple concurrency levels are realized through the use of nested transactions. Thus, the granularity of parameters is the transaction. In the same way, we will demand that the granularity of parameters will be the nested database and not its contained data items. Therefore, there will be associated a parameter set with each nested database which applies to all that database's elements. Now, imagine a scenario where a transaction has created a nested database, NDB_A, with parameters different from the transaction's own write-parameters. If one allows the transaction to move its data items into NDB_A by only converting the type of the lock from write to NDB, then NDB_A would contain a data item with parameters different from the parameters of NDB_A. Hence, violating the above stated rule about NDB parameter granularity. To prevent this, the parameters of data items being moved into a nested database are converted to the associated parameters of

the nested database.

5.5.2 Parameterized Access and Nested Databases

Let us take a look at how the integration of nested databases affects parameterized access. Figure 5.9 will guide us through the arguments. Recall that nested databases have associated write parameters and we assume that no restrictions are imposed on parameter usage.

In step 1, T_1 has write-locked a using the $W(B_{T_1})$ access mode and created a nested database NDB that uses $W(B_{NDB})$. In step 2, T_1 moves a into NDB. Now, this is actually a write parameter change on a's behalf. a's write parameters are changed from T_1's write parameters to NDB's write parameters: $W(B_{T_1}) \rightarrow W(B_{NDB})$. As with all write parameter modifications, if another transaction is reading a using $R(A)$ where $B_{T_1} \subseteq A$ and $B_{NDB} \not\subseteq A$, then this *implicit* parameter modification would also result in a conflict. This is not an explicitly requested modification of parameters and will therefore be referred to as *implicit parameter modifications.*

In step 3, T_2 visits NDB and write-locks a using $W(B_{T_2})$. Thus, a's write parameter set changes from $W(B_{NDB})$ to $W(B_{T_2})$. After possibly doing some updates to a, T_2 commits to NDB in step 4. Again, this represents an implicit parameter change, $W(B_{T_2}) \rightarrow W(B_{NDB})$, which could result in conflicts in the same way as pointed out above. In step 5, NDB commits and thereby changes the write parameters used to access a, back to B_{T_1}. Thus, resulting in another type of implicit parameter modification.

This analysis of parameter usage in NCCSR gives us the following three types of implicit write parameter modifications:

1. Moving data items into nested databases.
2. Committing visiting transactions of nested databases.
3. Committing nested databases.

5.5.3 View of parameters by reading visitors and observers

Consider step 4 in figure 5.9 again. Imagine that another transaction, say T_3, is visiting the global database using $R(A)$ and wants to observe the a data item. In order to test if this parameter access conflicts or not, which write parameters should be considered? Should T_3 relate to the write parameters of NDB (i.e. $W(B_{NDB})$), or to those of NDB's visiting transaction T_2 (i.e. $W(B_{T_2})$)? This question of behavior is addressed in [Anf97, page 52]:

> ...I tend to believe that such readers should simply see the parameter set of the DB lock. This is the simplest solution, and gives rise to more freedom inside subdatabase domains. Moreover, readers who need more detailed information about the reliability

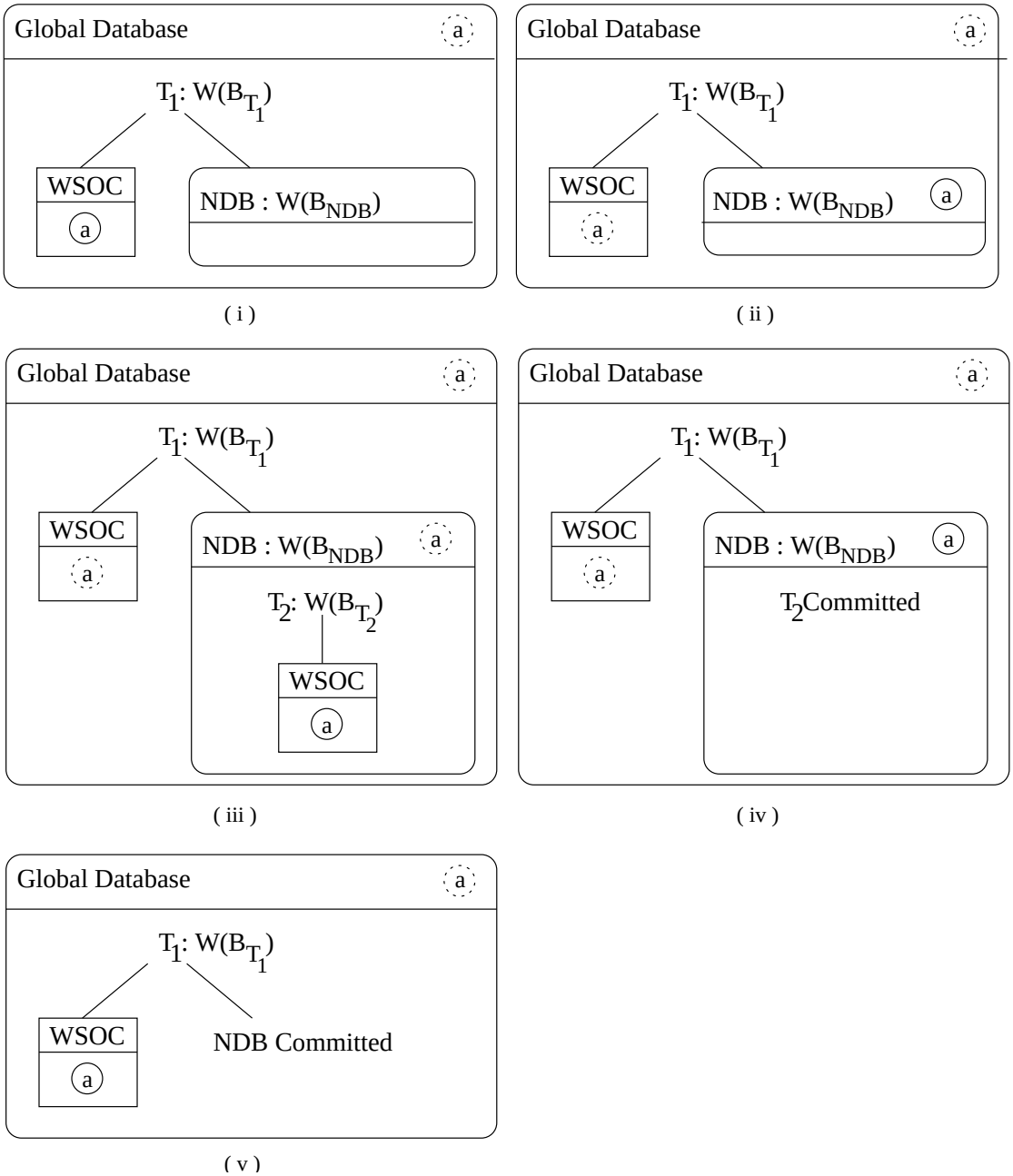


Figure 5.9: Parameterized nested databases.

of data items that are being manipulated inside a subdatabase, can establish a visiting transaction in the subdatabase in question.

By employing the strategy of letting observers see the nested database parameters, the implicit parameter modification of committing visiting transactions (type 2 above) would only affect other visiting transactions of the same nested database. Observers would always see the parameters of the nested database. In our example above, the observing transaction T_3 would see the parameters of NDB, $W(B_{NDB})$ and *not* those of T_2. On the other hand, if observers relate to the write parameters of visiting transactions, explicit and implicit modifications of those parameters would affect observing transactions in addition to visiting transactions.

5.5.4 Analysis of Explicit Modification of Parameters

Let us take a closer look at explicit modification of transaction and NDB parameters in context of NCCSR. We assume that an observing transaction sees the write parameters of the nested database from which it is observing and not those of any visiting transaction. Furthermore, we assume no restrictions on parameter usage by transactions and nested databases.

Modification of Read Parameters

Remember that a transaction can read data items by reading from the database it visits or by observing some descendant nested database. When modifying its read parameters the transaction, T_1 must

1. Avoid introducing conflicts with other visiting transactions that have write-locked data read by T_1.

2. Avoid introducing conflicts with nested databases from which it observes data.

Modification of Write Parameters

Only other visiting reading transactions can be involved in a conflict resulting from a write parameter modification. This is so because observing transactions only see the parameters of the nested database from which they observe and not from visiting writing transactions. Hence, a transaction, T, must

- Avoid introducing conflicts with other transactions visiting the same database and reading one or more data items from T.

Modification of NDB Parameters

The modification of nested database parameters can be used to communicate to observing transactions that the state of the uncommitted data contained by the database has changed. A nested database is a structuring mechanism

which does not have its own thread of execution, so it is the owner of the nested database that requests the parameter modification. When modifying NDB parameters the following has to be considered:

- Deal with introduced conflicts involving transactions observing the database.

5.5.5 Avoiding and Dealing with Conflicts in NCCSR

In the last sections we have analyzed the integration of CCSR and NCCSR in regard to parameterized accesses. As we know from section 5.4 conflicts can be introduced if parameters are modified. Read parameters can be expanded and write parameters can be restricted without causing conflicts, because these modifications only result in a lower degree of isolation. Section 5.5.2 taught us that parameters can not only be modified as a result of an explicit parameter modification request, but also implicitly. Following is a summary of all the actions that may result in parameters being modified. The first three actions are parameter modifications based on explicit requests, and the last three are implicit modifications.

1. Modification of read parameters
2. Modification of write parameters
3. Modification of NDB parameters
4. Moving data items into nested databases
5. Committing visiting transactions of nested databases
6. Committing nested databases

Let us take a look at how we might prevent these conflicts from being introduced by these actions and see how the techniques for conflict resolution from the pure CCSR case apply to the environment of NCCSR. Three strategies of increasing degree of parameter usage freedom will be given. The first aims at completely preventing conflicts by significantly restricting parameter usage. The second one only prevents denial of commit. Finally, the third does not impose any restrictions on parameter usage.

Strategy preventing all conflicts by restricting parameter usage

The basic idea here is to prevent conflicts by restricting the parameter usage to only allow loosening of isolation. This can be realized by ensuring that all of the above six parameter modification actions only can result in less isolation. For the actions 1 through 3 this looks easy. Read parameters can only be expanded, and write and NDB parameters can only be restricted.

Recall that moving a data item into a nested database (action 4) results in the conversion of the data item's write parameters to the parameters of the nested database. Therefore, we demand that the parameter values of a nested database have to be equal to or a subset of the write parameters of the transaction that owns it: $B_{NDB} \subseteq B_T$. This ensures that the write parameters of a data item being moved into a nested database only can be restricted. Imagine now a nested database, NDB, using B_{NDB} for its parameters, that the transaction, T, that owns it uses $W(B_T)$, and $B_{NDB} \subset B_T$. T moves one of its data items, x, into NDB. This results in a restriction of the write parameters of x. So far, so good. But, when T decides to commit NDB (action 6), the parameters of x are converted back to B_T. This conversion can introduce conflicts and can not be allowed. To assure that the commitment of nested databases does not lead to expanded write parameter sets, the owning transaction must use write parameters that are equal to or more restricted than those of its nested database. By combining this demand with the previous one we get $B_T \subseteq B_{NDB} \wedge B_{NDB} \subseteq B_T$, which is the same as $B_T = B_{NDB}$. Thus, a transaction can only create a database with the same parameters as its write parameters. This also leads to the operation of modifying NDB parameters only being allowed if the write parameters of the owner also are changed to be the same as the new NDB parameters. In the same way, visiting transactions can only use write parameters that are equal to or a superset of the database they visit. They are not allowed to restrict their write parameters to a level where they are more restrictive than the NDB parameters. This ensures that when they commit (action 5), the write parameters of their write-locked items are not modified in a manner that could introduce conflicts. These are significant restrictions of parameter use which deprive transactions and nested databases of their freedom of parameter usage.

Strategy preventing denial of commit (action 5 and 6)

It is desirable that transactions and nested databases are able to commit when they are ready. They should not be given the burden of sorting out introduced conflicts. This is particularly important in environments where the transactions represent the work of a user. When the user has completed his work and wants to commit it, he assumes that this should be painless (e.g. like saving a document).

This approach tries to prevent conflicts being introduced by actions 5 and 6. Transactions are only allowed to move data items into a nested database if this does not introduce any conflict. Thus, a transaction can be denied moving a data item into a subdatabase. The transaction could for example be given the possibility to resolve the conflict by forcing the conflicting reader(s) to change their parameters by using one of the strategies from section 5.4.3. Given our obtained knowledge from the previous approach, we eliminate the

conflicts of action 5 and 6 by requiring:

1. A transaction's write parameters must be equal to or a subset of the parameters all its nested databases.

2. A nested database can only use parameters that are a superset of its owner's write parameters.

3. A visiting transaction can only use write parameters that are a superset of the parameters of the database it visits.

From these rules we see that both nested databases and visiting transactions are allowed to both expand and restrict their write parameters as long as the rules hold. The expansion of write and NDB parameters can introduce conflicts with reading transactions. In the case expansion of NDB parameters, only observing transactions are involved in the conflict and in the case of transaction write-parameters, only transactions that visit the same nested database are involved. When modifying a transaction's write parameters one could use one of strategies from section 5.4.3 to resolve conflicts. But, when modifying nested database parameters theses techniques do not apply. They only deal with transactions, not nested databases. One way to resolve the conflict would be to use one of the techniques on the observers involved. The observers could be forced to sufficiently expand their read parameters or maybe split their transaction. As with modification of write parameters, conflicts resulting from modification of read parameters could be dealt with by the strategies from CCSR.

Strategy for maximizing freedom of parameter usage

This strategy gives the maximum amount of freedom by not imposing restrictions on parameter usage. The disadvantage is that the transactions and nested databases have to resolve any conflicts introduced by their commitment. Recall that the transactions preventing the commit are transactions reading data items with read parameters that would be incompatible with the write parameters of the items after the commit. One could resolve these conflicts by forcing the readers to change their read parameters on the items by using one of the strategies presented in section 5.4.3.

5.5.6 Summary

We have analyzed the consequences that the integration of CCSR with NCSR has on dynamic parameter modification. We saw that in addition to the issues of explicit requested parameter modification request, three other actions resulted in implicit parameter modification. The parameters of nested database were defined to apply to all data items contained by the database, and observing transactions see the NDB parameters of the databases they are

observing and not those of any visiting transactions within the observed database.

Three strategies where introduced to, in varying degree, minimize the appearance of conflicts due to implicit and explicit parameter modifications. Depending on how different application domains would use transaction and nested databases parameters and in what way they would be dynamically modified, one of the suggested strategies could be used. The following is a brief example of NCCSR parameter usage in the domain of software engineering.

Example. Let us return to Linda and Bill, the software engineers. Bill has been working on some source files inside the nested database of Linda. He is given the responsibility of integrating some files into, say, a reusable component. The transaction he currently is running uses $W(medium)$. He creates a nested database, NDB_{Bill}, using *low* for the NDB parameters to indicate the reliability of the component. Then, he moves the involved files into his nested database. This represents an implicit parameter modification and any resulting conflicts have to be resolved. In this application domain it would be reasonable to expect the readers involved in the conflict to yield. They could be requested to change their read parameters by one of the strategies of section 5.4.3. Bill now gives access privileges to the members of the team that will be working on the task. They start working on the files with *low* parameters and upgrade the parameters as the reliability increases. By using the parameter mappings of section 5.4.3 each upgrade would result in a restriction of the write parameters. Bill, could change the NDB parameters as the combined reliability of the task increases. Finally, when the task is completed, he commits NDB_{Bill} and Linda accepts, rejects, or denies the commit. The parameters of the data items are now converted back to those of Bill's transaction. Recall, that his transaction used $W(medium)$ at the time of creation. Note, that the data items' correct reliability state is *high* so it would not be right to convert them to *medium*. Bill could prevent this by upgrading his write parameters to *high* before the commit of the NDB_{Bill} or the system could prevent the situation from occurring by enforcing the rule of the strategy preventing denial of commit above, which only allows NDB_{Bill} to use parameters that are equal to or a superset of the write parameters of Bill's transaction. To follow the rule Bill would have to upgrade his transaction's write parameters each time he upgraded NDB_{Bill}'s parameters.

5.6 Summary

This chapter has analyzed the issues concerning dynamic modification of isolation in the contexts of both CCSR and NCCSR. We first analyzed the

less complex CCSR case and suggested a number of strategies for dealing with conflicts introduced by read and write parameter modifications. Then, we integrated CCSR with NCSR and analyzed the consequences in regard to read, write, and NDB parameter modifications. Finally, three strategies of increasing degree of parameter usage freedom where suggested and discussed.

Chapter 6

Conclusions and Future Work

6.1 Introduction

This chapter gives an evaluation of the results and a summary of the contributions of this thesis. It also presents two areas for future work.

6.2 Evaluation of Results

The previous chapter showed how a parameter modification can introduce conflicts with transactions that use conflicting parameters. A number of strategies to deal with these conflicts weere suggested. The strategies that aim at resolving conflicts can be used in two ways. They can be used in a way that only affects the transaction requesting the parameter modification, and they can be used in a way that affects the other transactions involved in the conflict. An example of the latter was given on page 78 where Bill downgraded the reliability of his work to *low* and thereby forced the observing transaction of Kate accepting only *medium* reliability to resolve the created conflict. This leads to discussion of the semantics of a parameterized lock. If a parameterized read-lock is held on a data item using e.g. $R\{medium\}$, should this mean that the transaction is granted the right to read the data item using $R\{medium\}$ till it decides to release the lock? Given these semantics, Bill would have no right to force Kate to resolve the conflict because she has been granted the right to read the data using the *medium* parameter. Another way to define the semantics of a parameterized lock would be to state that if a transaction holds a read-lock using some parameters it is granted read access to the locked data. However, it does not have locked the right to use the associated parameters. This means that the transaction could be requested to modify its parameters to avoid conflicts with other transactions. These semantics would allow the actions in the example above.

Both the mentioned semantics are useful. The first definition would be useful to transactions whose execution depends on the ability to access the

locked data using the associated parameters. If they were forced to change their parameters, they would not be able to continue and would thus have to abort. The second definition is useful in e.g. collaborative environments where there exist transactions that observe some uncommitted data knowing that the level of reliability of the data could change to an undesirable level. Here it should not be the writers problem if some reader does not accept the new parameters. This is the situation of most collaborative application domains, because collaboration is often enabled by relaxing isolation and thus letting transactions read uncommitted data.

Due to the value of the two semantics of parameterize locks described, it would be desirable if a CCSR transaction manager supported both. One could imagine the implementation of two types of locks. The first supports locked parameterized access (the first described lock semantics above) and the second could allow a configurable policy which dictates how the associated parameters are allowed to be modified by other transactions.

Allowing dynamic modification of parameters introduces conflicts under circumstances pointed out in the previous chapter. These conflicts have to be avoided or resolved. Which methods that could be used to achieve this depends on how the application uses parameterized access and nested databases provided by Apotram.

6.3 Contributions of this Thesis

One of the requirements of the Apotram transaction model is the ability to dynamically modify a transaction's concurrency level [Anf97, page 55]. This is realized by allowing transactions to dynamically modify their parameterized access modes. The analysis of consequences of allowing dynamic modification was not within the scope of O.J. Anfindsen's thesis and was therefore left to further research. This thesis has analyzed the implications of dynamic parameter modification. First, dynamical modification of parameters was analyzed in the context of *conditional conflict serializability* (CCSR) only and then the focus was shifted to the more complex context of *nested conditional conflict serializability* (NCCSR). In both cases it was pointed out under what circumstances parameter modifications result in conflicts. In addition, a number of strategies to deal with discovered issues were suggested. These strategies can be classified into two categories, those that prevent conflicts from being introduced and those that provide a mechanism to remove introduced conflicts.

The above stated contributions are summarized below.

- Analysis of dynamic parameter modifications under CCSR and a discussion of the consequences.

- Analysis of dynamic parameter modifications under NCCSR and a discussion of the consequences.
- Suggested strategies for avoiding and resolving conflicts under CCSR.
- Suggested strategies for avoiding and resolving conflicts under NCCSR.

These contributions are valuable when implementing and using Apotram with support for dynamic parameter modification.

6.4 Possible Future Work

6.4.1 Integration of Apotram and Split/Join Operations

A important contribution to transaction models is the concept of dynamic restructuring of in-progress transactions suggested in [Kai95] (and briefly summarized in this thesis on page 47). Some of the strategies to deal with dynamic parameter modification presented in the previous chapter used the split and split-commit operations to resolve conflicts. To use these strategies it is therefore necessary to integrate Apotram with the operations that provide restructuring of transactions. The additional functionality provided to Apotram could enhance Apotram in other ways and has to be further investigated, as stated in [Anf97, page 108].

6.4.2 Case Study giving a Practical Example of Parameter Modification

Initially this thesis was supposed to include a chapter giving a case study describing how a concrete application domain that requires support for collaboration and long-lived transactions could be implemented using Apotram. It would be investigated how the requirements could be met by the use of Apotram and dynamic parameter modification. This case study would provide a concrete example of practical use of Apotram and its support for dynamic modification of parameters. Unfortunately, it was not possible to conduct it within the time-frame of the master thesis.

Such a case study would be a valuable contribution by showing how real-world applications could use and benefit from dynamic parameter modification.

List of Figures

Bibliography

[AA92] D. Agrawal and A. El Abbadi. *Transaction Management in Database Systems*, chapter 1, pages 1–31. In Elmagarmid [Elm92], 1992.

[Anf97] Ole Jørgen Anfindsen. *Apotram - an Application-Oriented Transaction Model.* PhD thesis, Department of Informatics, University of Oslo, Norway, March 1997. Available online at http://www.apotram.com.

[Anf00a] Ole Jørgen Anfindsen. Apotram – Technical summary. Available online at http://www.apotram.com, 2000.

[Anf00b] Ole Jørgen Anfindsen. Collaborative transactions. Telenor Research and Development / Apotram AS, Available online at http://www.apotram.com, July 2000.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1987.

[BN97] Philip A. Bernstein and Eric Newcomer. *Principles of Transaction Processing.* Morgan Kaufmann Publishers, 1997.

[DA00a] Laurent Daynès and Ole Jørgen Anfindsen. Implementation of Apotram Nested Databases using Flexible Locking Mechanisms. Document SML-2000-0206, Sun Microsystems Laboratories, Mountain View, California, US, July 2000.

[DA00b] Laurent Daynès and Ole Jørgen Anfindsen. Implementation of Parameterized Lock Modes using Ignore-Conflict Relationships. Document SML-99-0602, Sun Microsystems Laboratories, Mountain View, California, US, July 2000.

[Dav78] C. T. Davies, Jr. Data processing spheres of control. *IBM Systems Journal*, 17(2):179–198, 1978.

[Elm92] Ahmed K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, 1992.

[ELMB92] Ahmed K. Elmagarmid, Yungho Leu, James G. Mullen, and Omran Bukhres. *Introduction to Advanced Transaction Models*, chapter 2, pages 33–52. In Elmagarmid [Elm92], 1992.

[EN94] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems.* Addison Wesley, second edition, 1994.

[Eva00] Huw Evans. Apotram - Its Applicability to Run-time Software Evolution. Department of Computer Science, The University of Glasgow, UK, February 2000.

[GMS87] Hector Garcia-Molina and Kenneth Salem. SAGAS. In *Proceedings of ACM SIGMOD International Conference*, pages 249–259, 1987.

[GR93] Jim Gray and Andreas Reuter. *Transaction processing: concepts and techniques.* Morgan Kaufmann, 1993.

[Gra81] Jim Gray. The transaction concept: Virtues and limitations (invited paper). In *International Conference On Very Large Data Bases (VLDB '81)*, pages 144–154, Los Angeles, Ca., USA, September 1981. IEEE Computer Society Press.

[Gri94] Ralph P. Grimaldi. *Discrete and combinatorial mathematics - An applied introduction.* Addison Wesley, 3rd edition, 1994.

[HR83] Theo Haerder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, dec 1983.

[JK97] Sushil Jajodia and Larry Kerschberg, editors. *Advanced Transaction Models and Architectures.* Kluwer, 1997.

[JSGB00] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java Language Specification.* The Java Series. Addison-Wesley, second edition, June 2000. Available online at http://java.sun.com/docs/books/jls.

[Kai95] Gail E. Kaiser. *Cooperative Transactions for Multiuser Environments*, chapter 20, pages 409–433. In Kim [Kim95], 1995.

[Kim95] Won Kim, editor. *Modern Database Systems.* Addison Wesley, 1995.

[KP92] Gail E. Kaiser and Carlton Pu. *Dynamic Restructuring of Transactions*, chapter 8, pages 266–295. In Elmagarmid [Elm92], 1992.

[Mos81] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* PhD thesis, MIT Dept. of Elec. Eng. and Comp. Sci., April 1981.

[Mos85] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* MIT Press, 1985.

[SKS97] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts.* McGraw-Hill, third edition, 1997.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, 1992.

[US92] Rainer Unland and Gunter Schlageter. *A Transaction Manager Development Facility for Non Standard Database Systems*, chapter 11, pages 399–466. In Elmagarmid [Elm92], 1992.